Model-Based Testing with Graph Grammars

MSc Thesis *(Afstudeerscriptie)*

written by

Vincent de Bruijn

Formal Methods & Tools,
University of Twente,
Enschede,
The Netherlands

v.debruijn@student.utwente.nl

January 11, 2013

Abstract

Graph Grammars describe system behavior through graphs and graph transformation rules. Graph Grammars have not been used for Model-Based Testing. However, Graph Grammars have many structural advantages, which are potential benefits for the model-based testing process. We describe a model-based testing setup with Graph Grammars. The result is a system for automatic test generation from Graph Grammars. A graph transformation tool, GROOVE, and a model-based testing tool, ATM, are used as the backbone of the system. The system is validated using the results of several case studies.

The result of this report is a tool, GRATiS, which can be used for model-based testing on Graph Grammars.

hier komt ook nog een kort overzicht van de conclusies

Contents

Chapter 1

Introduction

In this introduction, first the importance of testing and automation of testing is stressed. Then Model-Based Testing is shown to be a useful tool for automation of testing. Graph Grammars and graph transformation are argued to be useful as formalism for Model-Based Testing. Some leading tools for automatic test generation are set out, which include the tools used in this report. The research goals are given and finally a roadmap explains the basic structure of the rest of this report.

1.1 Testing

In software development projects, often time and budget costs are exceeded. Laird and Brennan [11] investigated in 2006 that 23% of all software projects are canceled before completion. Furthermore, of the completed projects, only 28% are delivered on time with the average project overrunning the budget with 45%. The cause of this often are the unclear ambigious requirements of the software system to develop.

Testing is an important part of software development, because it decreases future maintainance costs [17]. Testing is a complex process and should be done often [20]. Therefore, the testing process should be as efficient as possible in order to save resources.

Test automation allows repeated testing during the development process. The advantage of this is that bugs are found early and can therefore be fixed early. A widely used practice is maintaining a *test suite*, which is a collection of test-cases. However, when the creation of a test suite is done manually, this still leaves room for human error [14]. The process of deriving tests tends to be unstructured, barely motivated in the details, not reproducible, not documented, and bound to the ingenuity of single engineers [28].

1.2 Model-based Testing

The existence of an artifact that explicitly encodes the intended behaviour can help mitigate the implications of these problems. Creating an abstract representation or a *model* of the system is an example of such an artifact. What is meant by a model in this report, is the description of the behavior of a system. In particular, the term model will be often used to describe transition-based notations, such as finite state machines, labelled transition systems and I/O automata. Other notations, such as UML statecharts, are not considered as models in this report.

A model can be used to systematically generate tests for the system. This is referred to as *model-based testing*. Generating tests automatically leads to a larger test suite than if done manually. A large, systematically built test suite is bound to find more bugs than a smaller, manually built one.

Models are created from the specification documents provided by the end-user. These specification documents are 'notoriously error-prone' [16]. This implies that the model itself needs validation. Validating the model usually means that the requirements themselves are scrutinised for consistency and completeness [28]. This helps to clear up ambigious requirements early on, which allows better estimation of the budget and time demands.

The stakeholders evaluate the constructed model to verify its correctness. However, the visual or textual representation of large models may become troublesome to understand, which is referred to as the model having a low model transparency or high model complexity. The problem with transition systems is that a larger number of states and/or transitions decreases the model transparency. We think that low model transparency make errors harder to detect and that it obstructs the feedback process of the stakeholders. Using models with high transparency is therefore essential.

1.3 Graph Transformation

A formalism that claims to have more model transparency is Graph Transformation. The system states are represented by graphs and the transitions between the states are accomplished by applying graph change rules to those graphs. These rules can be expressed as graphs themselves. A graph transformation model of a software system is therefore a collection of graphs, each a visual representation of one aspect of the system. This formalism may therefore provide a more intuitive approach to system modelling than traditional state machines. Graph Transformation and its potential benefits have been studied since the early '70s. The usage of this computational paradigm is best described by the following quote from Andries et al. [1]:

> Graphs are well-known, well-understood, and frequently used means to represent system states, complex objects, diagrams, and networks, like flowcharts, entity-relationship diagrams, Petri nets, and many more. Rules have proved to be extremely useful for describing computations by local transformations: Arithmetic, syntactic, and deduction rules are well-known examples.

An informative paper on graph transformations is written by Heckel et al. [9]. A quote from this paper:

> Graphs and diagrams provide a simple and powerful approach to a variety of problems that are typical to computer science in general, and software engineering in particular.

1.4 Tools

Tools for automatic test generation already exist. In Utting et al. [28], a taxonomy is done on different model-based testing tools:

- TorX [25]: accepts behaviour models such as I/O labelled transition systems. A version of this tool written in Java under continuous development is JTorX [2]. This version accepts the same kind of models as ATM.
- Spec Explorer[29]: provides a model editing, composition, exploration and visualization environment within Visual Studio, and can generate offline .NET test suites or execute tests as they are generated (online).

- JUMBL[21]: an academic model-based statistical testing that supports the development of statistical usage-based models using Markov chains, the analysis of models, and the generation of test cases.
- AETG[5]: implements combinatorial testing, where the number of possible combinations of input variables are reduced to a few 'representative' ones.
- STG tool[4]: implements conformance testing techniques to automatically derive symbolic test cases from formal operational specifications.

The testing tool developed by Axini[1] is used for the automatic test generation on *symbolic* models, which combine a state and data type oriented approach. This tool is referred to as Axini Test Manager (ATM) and is used in practice by several Dutch companies.

The choice was made to use ATM, because of the location of Axini, their willingness to support this project and the already available models for case studies. Another interesting option for our research would have been to use an open-source MBT tool. In particular JTorX was an interesting candidate due to its maturity and the available support at the University of Twente.

The graph transformation tool GROOVE[2] is used to model and explore graph grammars.

Zijn er nog andere graph transformation tools?

1.5 Research goals

The motivation above is given for using graph grammars as a modelling technique. The goal of this research is to create a system for automatic test generation on graph grammars. If the assumptions that graph grammars provide a more intuitive modelling and testing process hold, this new testing approach will lead to a more efficient testing process and fewer incorrect models. The system to be designed, once implemented and validated, should provide a valuable contribution to the testing paradigm. The tools GROOVE and ATM are used to create this system.

The research goals are split into a design and validation component:

1. **Design**: Design and implement a system using ATM and GROOVE which performs model-based testing on graph grammars.
2. **Validation**: Validate the design and implementation using case studies and performance measurements.

The result of the design goal is one system called the GROOVE-Axini Testing System (GRATiS). The validation goal uses case-studies with existing specifications from systems tested by Axini. Each case-study has a graph grammar and a symbolic model which describe the same system. GRATiS and ATM are used for the automatic test generation on these models respectively. Both the models and the test processes are compared as part of the validation.

The solution has to uphold three requirements:

1. A graph grammar must be used as the model; it must derive from the specification and be used for the testing.
2. It must be possible to measure the test progress/completion, by means of *coverage* statistics (explained in detail in section 2.1.2).
3. The solution must be efficient: it should be usable in practice, therefore the technique should be scalable and the imposed constraints reasonable from a practical view point.

[1] http://www.axini.nl/

[2] http://sourceforge.net/projects/groove/

1.6 Roadmap

This report has five more chapters: first, the concepts described in this chapter are elaborated in chapter 2. The design of GRATiS is described in chapter 3. The implementation of GRATiS is covered in chapter 4. The validation of GRATiS is in chapter 5. Finally, conclusions are drawn in chapter **??**.

hoofdstukken met ?? zitten niet in dit verslag

Chapter 2

Background

The structure of this chapter is as follows: the general model-based testing process is set out in section 2.1. Some basic concepts from algebra are described in section 2.2. Then the symbolic models used by ATM are described in section 2.3. Section 2.4 describes the Graph Grammar formalism. GROOVE and ATM are described in section 2.5.

2.1 Model-based Testing

Model-based testing is a testing technique where a System Under Test (SUT) is tested for conformance to a model description of the system. The general setup for this process is depicted as a UML sequence diagram in Figure 2.1. The specification of a system is provided as a model to a test derivation component which generates a test suite. The test suite is used by a test execution component to test the SUT. Tests are executed by providing input/stimuli to the SUT and monitoring the output/response. The test execution component evaluates whether the correct responses are given. It gives a 'pass' or 'fail' verdict depending on whether the SUT conforms to the model or not.

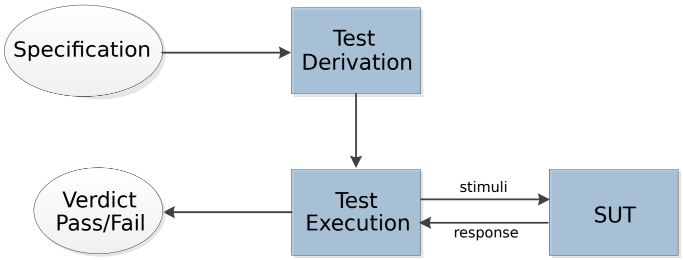


Figure 2.1: A general model-based testing setup

This type of model-based testing is called *batch testing* or *offline testing*. Another type of model-based testing is *on-the-fly* testing. The main difference is that no test cases are derived, instead a transition in the model is chosen and tested on the system directly. The general architecture for this process is shown in Figure 2.2. An example of an on-the-fly testing tool is TorX [25].

Variations of state machines and transition systems have been widely used as the underlying model for test generation. Other tools use the structure of data types to generate test data.

The stucture of the rest of this section is as follows. First, previous work on model-based testing is given. Then, Labelled Transition Systems are introduced. This is a basic formalism useful to

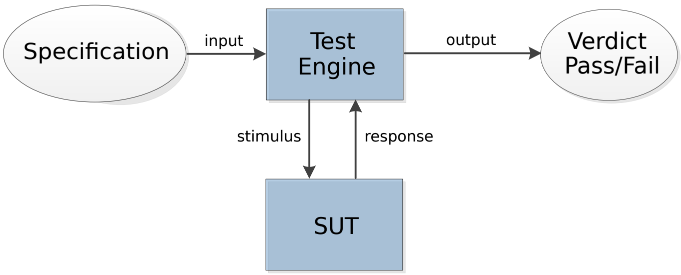


Figure 2.2: A general 'on-the-fly' model-based testing setup

understand the models in the rest of the paper, namely Graph Grammars and Symbolic Transition Systems. Next, an adaptation of Labelled Transition Systems, the Input-Output Transition System is described. This is a useful formalism for Model-Based Testing. Finally, the notion of *coverage* is explained.

2.1.1 Previous work

Transition-based formal testing theory was introduced by De Nicola et al. [19]. The input-output behavior of processes is investigated by series of tests. Two processes are considered equivalent if they pass exactly the same set of tests. This testing theory was first used in algorithms for automatic test generation by Brinksma [3]. This led to the so-called *canonical tester* theory. Tretmans gives a formal approach to protocol conformance testing (whether a protocol conforms to its specifications) in [26] and an algorithm for deriving a sound and exhaustive test suite from a specification in [27]. A good overview of model-based testing theory and past research is given in "Model-Based Testing of Reactive Systems" [15].

2.1.2 Labelled Transition Systems

A Labelled Transition System (LTS) is a structure consisting of states with labelled transitions between them.

Definition 2.1.1. Labelled Transition Systems
An LTS is a 4-tuple $\langle Q, L, T, q_0 \rangle$, where:

- Q is a finite, non-empty set of states
- L is a finite set of labels
- $T \subseteq Q \times (L \cup \{\tau\}) \times Q$, with $\tau \notin L$, is the transition relation
- $q_0 \in Q$ is the initial state.

We write $q \xrightarrow{\mu} q'$ if there is a transition labelled μ from state q to state q', i.e., $(q, \mu, q') \in T$. q, q' are called the source and target states of the transition respectively. The informal idea of such a transition is that when the transition system is in state q it may perform action μ, and go to state q'.

Input-Output Transition Systems A useful type of transition system for model-based testing is the Input-Output Transition System (IOTS) by Tretmans [27]. Assuming that implementations communicate with their environment via inputs and outputs, this formalism is useful for describing system behavior. The system is regarded as a black box and the IOTS specifies the allowed inputs and outputs.

IOTSs have the same definition as LTSs with one addition: each label $l \in L$ has a type $\iota \in Y$, where $Y = \{input, output\}$. Each label can therefore specify whether the action represented by the label is a possible input or an expected output of the system under test. An IOTS is formally defined as:

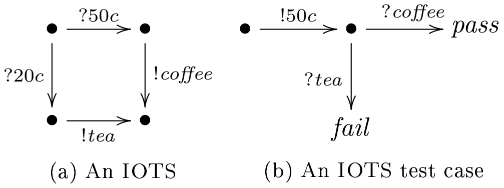
Definition 2.1.2. Input-Output Transition Systems
An IOTS is a 4-tuple $\langle Q, L, T_Y, q_0 \rangle$, where $T_Y \subseteq T \times Y$ are the input-output transitions.

When the transition system is in the source state of an input transition, the input can be given to the SUT. When the transition system is in the source state of an output transition, the output should be observed from the SUT. In both cases, the transition system advances to the target state of the transition. The case where a state has both input and output transitions is not considered in this report.

An example of such an IOTS is shown in Figure 2.3a. This system allows an input of 20 or 50 cents and then outputs tea or coffee accordingly. The inputs are preceded by a '?', the outputs are preceded by an '!'. This system is a specification of a coffee machine. A test case can also be described by an IOTS with special pass and fail states.

A test case for the coffee machine is given in Figure 2.3b. The test case shows that when an input of '50c' is given, an output of 'coffee' is expected from the tested system, as this results in a 'pass' verdict. When the system responds with 'tea', the test case results in a 'fail' verdict. The pass and fail verdicts are two special states in the test case, which are sink states, i.e., once in either of those the test case cannot leave that state.

Test cases should always reach a pass or fail state within finite time. This requirement ensures that the testing process halts.



(a) An IOTS (b) An IOTS test case

Figure 2.3: The specification of a coffee machine and a test case as an IOTS

Coverage The number of tests that can be generated from a model is potentially infinite. Therefore, there must be a test selection strategy to maximize the quality of the tests while minimizing the time spent testing. Coverage statistics help with test selection. Such statistics indicate how much of the model is tested. A transition in the model is considered tested when the transition system is in the source state of the transition and either the input is given to or the output is observed from the SUT, depending on the type of the transition.

When the SUT is a black box, typical coverage metrics are state and transition coverage of the model [13, 18, 8]. State coverage can be measured by dividing the tested states by the total states in the model. The same process applies to transition coverage. As an example, the coverage metrics of the IOTS test case example in 2.3b are calculated. The test case tests one path through the specification and passes through 3 out of 4 states and 2 out of 4 transitions. The state coverage is therefore 75% and the transition coverage is 50%.

Coverage statistics are calculated to indicate how adequately the testing has been performed [30]. These statisics are therefore useful metrics for communicating how much of a system is tested.

2.2 Algebra

Some basic concepts from algebra are described here. For a general introduction into logic we refer to [10]. This section explains in order: multi-sorted signatures, algebrae, variables & terms and term-mapping & valuations. The algebra described here will be used in the next sections to formally define Symbolic Transition Systems and Graph Grammars.

Definition 2.2.1. Multi-sorted Signatures
A *multi-sorted signature* $\langle S, F\rangle$ describes the sorts and function symbols of a formal language. S is a set of sorts. F is a set of function symbols. Each $f \in F$ has an arity $n \in \mathbb{N}$, where a function symbol with arity $n = 0$ is called a constant symbol. F^i denotes the subset of F with function symbols of arity $n = i$. The sort of a function symbol $f \in F$ with arity n is given by $\sigma(f) = s_1 ... s_{n+1}$, with $s_i \in S$ for $1 \leq i \leq n$. s_{n+1} is the return sort. In this report, $S = \{int, real, bool, string\}$ denoting the integer, real, boolean and string sorts respectively. F features the commonly used function symbols, which include, but not restricted by, '+', '*', '=', '<', '0', '1'.

Definition 2.2.2. Algebrae
An *algebra* $\mathcal{A} = \langle \mathbb{U}, \Phi\rangle$ has a non-empty set $\mathbb{U}$ of values called a *universe*, partitioned into $\mathbb{U}^s$ for each $s \in S$, and a set Φ of functions. A function $\phi_{\mathcal{A}}$ is typed $\mathbb{U}^{s_1}_{\mathcal{A}} \times ... \mathbb{U}^{s_n}_{\mathcal{A}} \rightarrow \mathbb{U}^{s_{n+1}}_{\mathcal{A}}$, where $s_1 ... s_{n+1}$ is the sort of the function symbol given by the signature. For example, $<_{\mathcal{A}}$: $\mathbb{U}^{int}_{\mathcal{A}} \times \mathbb{U}^{int}_{\mathcal{A}} \rightarrow \mathbb{U}^{bool}_{\mathcal{A}}$ represents the 'less-than' comparison of two integers.

Definition 2.2.3. Point algebra
We define a *point algebra* $\mathcal{P}$ to be an algebra with $\forall s \in S . |\mathbb{U}^s_{\mathcal{P}}| = 1$.

Definition 2.2.4. Variables
We define $\mathcal{V} = \mathcal{V}^{int} \uplus \mathcal{V}^{real} \uplus \mathcal{V}^{bool} \uplus \mathcal{V}^{string}$ to be the set of *variables. Terms* over V, denoted $\mathcal{T}(V)$, are built from function symbols F and variables $V \subseteq \mathcal{V}$. The definition of a term is:

$$\begin{array}{lll} t & ::= & f(t_1 ... t_n) \quad , \textit{where}\ \mathrm{n}\ \textit{is the arity of}\ \phi \\ & | & x \qquad\qquad\quad , \textit{where}\ x\ \textit{is a variable.} \end{array}$$

We write $var(t)$ to denote the set of variables appearing in a term $t \in \mathcal{T}(V)$. Terms $t \in \mathcal{T}(\emptyset)$ are called ground terms. An example of a term t is $(x + (y - 1))$, with $var(t) = \{x, y\}$. The type of a term is given by:

$$\begin{array}{lll} \sigma : t \mapsto & s & \textit{if}\ t = x \in \mathcal{V}^s \\ & s_{n+1} & \textit{if}\ t = f(t_1 ... t_n)\ \textit{and}\ \sigma(f) = s_1 ... s_{n+1},\ \textit{provided}\ \sigma(t_i) = s_i \end{array}$$

The set of terms with return type *bool*, is denoted as $\mathcal{B}(\mathcal{V})$. An example is $(x < y)$, where the result is *true* or *false*.

Definition 2.2.5. Term-mapping
A *term-mapping* is a function $\mu : \mathcal{V} \rightarrow \mathcal{T}(\mathcal{V})$. A *valuation* ν is a function $\nu : \mathcal{V} \rightarrow \mathbb{U}$ that assigns values to variables. For example, given an algebra, $\nu : \{(x \mapsto 1), (y \mapsto 2)\}$ assigns the values 1 and 2 to the variables x and y respectively. A valuation of a term given $\mathcal{A}$ is defined by:

$$\begin{array}{lll} \nu : & x & \mapsto \quad \nu(x) \\ & f(t_1 ... t_n) & \mapsto \quad f_{\mathcal{A}}(\nu(t_1) ... \nu(t_n)) \end{array}$$

When every variable in a term is defined by a valuation, the term can be valuated to a value. Therefore, when every variable in a term-mapping is defined by a valuation, a new valuation can be obtained. Formally, this is defined as: $\_after\_ : (\mathcal{V} \rightarrow \mathbb{U}) \times (\mathcal{V} \rightarrow \mathcal{T}(\mathcal{V})) \rightarrow (\mathcal{V} \rightarrow \mathbb{U})$. Given a valuation ν and a term-mapping μ, $(\nu\ after\ \mu) : \nu \mapsto \nu(\mu(\nu))$.

2.3 Symbolic Transition Systems

Symbolic Transition Systems (STSs) combine a state oriented and data type oriented approach. This formalism is a specification of system behavior like LTSs. These systems are used in practice in ATM and will therefore be part of GRATiS. In this section, previous work on STSs is reviewed. The definitions of STSs and IOSTSs follow. An example of an IOSTS is then given. Next, the mapping of an STS to an LTS is explained and illustrated by an example. This mapping is useful when comparing STSs to Graph Grammars, because both systems can be mapped to an LTS and then compared. Finally, different coverage metrics on STSs are explained.

2.3.1 Previous work

STSs are introduced by Frantzen et al. [12]. This paper includes a detailed definition, on which the definition below is based. The authors also give a sound and complete test derivation algorithm from specifications expressed as STSs. Deriving tests from a symbolic specification or *Symbolic test generation* is introduced by Rusu et al. [24]. Here, the authors use *Input-Output Symbolic Transition Systems* (IOSTSs). These systems are very similar to the STSs in [12]. However, the definition of IOSTSs we will use in this report is based on the STSs by [12]. A tool that generates tests based on symbolic specifications is the STG tool, described in Clarke et al. [4].

2.3.2 Definition

An STS has *locations* and *switch relations.* If the STS represents a model of a software system, a location in the STS represents a state of the system, not including data values. A switch relation defines the transition from one location to another. The *location variables* are a representation of the data values in the system. A switch relation has a *gate*, which is a label representating the execution steps of the system. Gates have *interaction variables*, which represent some input or output data value. Switch relations also have *guards* and *update mappings.* A guard is a term $t \in \mathcal{B}(\mathcal{V})$. The guard disallows using the switch relation when the valuation of the term results in *false.* When the valuation results in *true*, the switch relation of the guard is *enabled.* An update mapping is a term-mapping of location variables. After the system switches to a new location, the variables in the update mapping will have the value corresponding to the valuation of the term.

Definition 2.3.1. Symbolic Transition Systems
A Symbolic Transition System is a tuple $\langle W, w_0, \mathcal{L}, \imath, \mathcal{I}, \Lambda, D\rangle$, where:

- W is a finite set of locations.
- $w_0 \in W$ is the initial location.
- $\mathcal{L} \subseteq \mathcal{V}$ is a finite set of location variables.
- $\imath$ is a term-mapping $\mathcal{L} \to \mathcal{T}(\emptyset)$, representing the initialisation of the location variables.
- $\mathcal{I} \subseteq \mathcal{V}$ is a set of interaction variables, disjoint from $\mathcal{L}$.
- Λ is a finite set of gates. The unobservable gate is denoted $\tau(\tau \notin \Lambda)$; we write Λ_τ for $\Lambda \cup \{\tau\}$. The arity of a gate $\lambda \in \Lambda_\tau$, denoted $arity(\lambda)$, is a natural number. The parameters of a gate $\lambda \in \Lambda_\tau$, denoted $param(\lambda)$, are a tuple of length $arity(\lambda)$ of distinct interaction variables. We fix arity$(\tau) = 0$, i.e. the unobservable gate has no interaction variables.
- $D \subseteq W \times \Lambda_\tau \times \mathcal{B}(\mathcal{L} \cup \mathcal{I}) \times (\mathcal{L} \to \mathcal{T}(\mathcal{L} \cup \mathcal{I})) \times W$, is the switch relation. We write $w \xrightarrow{\lambda,\gamma,\rho} w'$ instead of $(w, \lambda, \gamma, \rho, w') \in D$, where γ is referred to as the guard and ρ as the update mapping. We require $(var(\gamma) \cup var(\rho)) \subseteq (\mathcal{L} \cup param(\lambda))$. We define $out(w) \subseteq D$ to be the outgoing switch relations from location w.

An IOSTS can now easily be defined. The same difference between the labels in LTSs and IOTSs apply, namely each gate has a type $\iota \in Y$. As with labels, each gate is preceded by an '?' or '!' to indicate whether it is an input or an output respectively. The full definition is as follows:

Definition 2.3.2. Input-Output Symbolic Transition Systems
An IOSTS is a 5-tuple $\langle W, \mathcal{L}, \imath, \Lambda_Y, D\rangle$, where $\Lambda_Y \subseteq \Lambda \times Y$ are the input-output gates.

2.3.3 Example

In Figure 2.4 the IOSTS of a simple board game is shown, where two players consecutively throw a die and move along four squares. The 'init' switch relation is a graphical representation of the variable initialization $\imath$. The values in the tuple of the IOSTS are defined as follows:

$$\begin{array}{lcl}
W & = & \{t, m\} \\
w_0 & = & t \\
\mathcal{L} & = & \{T, P1, P2, D\} \\
\imath & = & \{T \mapsto 0, P1 \mapsto 0, P2 \mapsto 2, D \mapsto 0\} \\
\mathcal{I} & = & \{d, p, l\} \\
\Lambda & = & \{?throw, !move\} \\
D & = & \{t \xrightarrow{?throw, 1<=d<=6, D\mapsto d} m, \\
 & & m \xrightarrow{!move, T=1 \wedge l=(P1+D)\%4, P1\mapsto l, T\mapsto 2} t, \\
 & & m \xrightarrow{!move, T=2 \wedge l=(P2+D)\%4, P2\mapsto l, T\mapsto 1} t\}
\end{array}$$

The variables $T, P1, P2$ and D are the location variables symbolizing the player's turn, the positions of the players and the number of the die thrown respectively. The output gate $!move$ has $param = \langle p, l\rangle$ symbolizing which player moves to which location. The input gate $?throws$ has $param = \langle d\rangle$ symbolizing which number is thrown by the die. The switch relation with gate $?throws$ has the restriction that the number of the die thrown is between one and six and the update sets the location variable D to the value of interaction variable d. The switch relations with gate $!move$ have the restriction that it must be the turn of the player moving and that the new location of the player is the number of steps ahead as thrown by the die. The update mapping sets the location of the player to the correct value and passes the turn to the next player. In Figure 2.4 the gates, guards and updates are separated by pipe symbols '|' respectively.

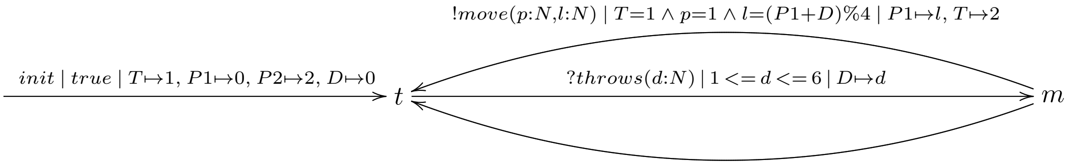

!move(p:N,l:N) | T=1 ∧ p=1 ∧ l=(P1+D)%4 | P1↦l, T↦2
init | true | T↦1, P1↦0, P2↦2, D↦0
t
?throws(d:N) | 1 <= d <= 6 | D↦d
m
!move(p:N,l:N) | T=2 ∧ p=2∧l=(P2+D) %4 | P2↦l, T↦1

Figure 2.4: The IOSTS of a board game example

2.3.4 STS to LTS mapping

This section shows the method of mapping an STS to an LTS.

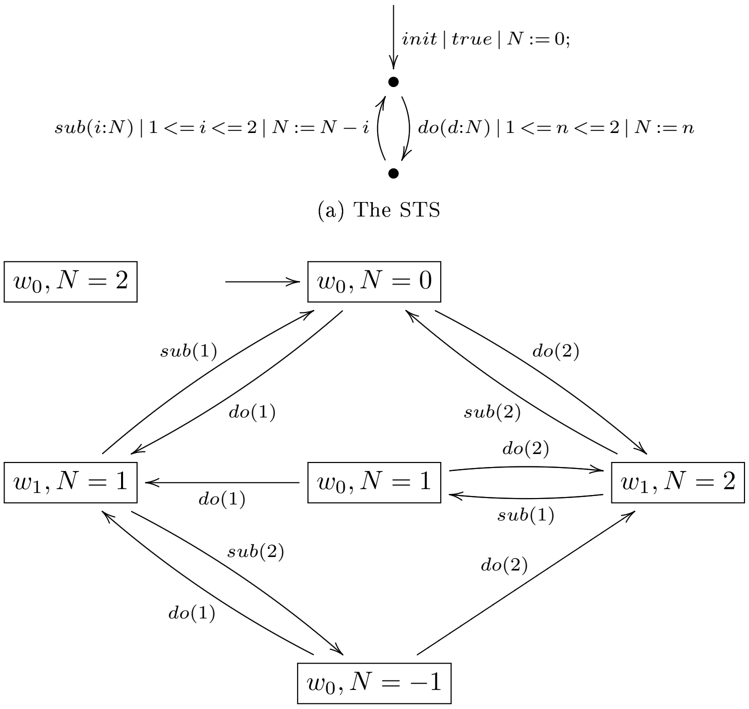
Consider an STS J. We can construct an LTS K from J, such that K is an expansion of J. There exists a mapping from the location and location variable valuations to the states of K and from the switch relations and variable valuations of J to the transitions of K. These relations are defined as follows:

Definition 2.3.3. STS to LTS mapping

$$\begin{aligned}
\mu_Q :&\quad (W \times (\mathcal{L} \to \mathbb{U})) \to Q\\
\mu_L :&\quad (\Lambda \times (\mathcal{I} \to \mathbb{U})) \to L\\
\mu_T :&\quad (w \xrightarrow{\lambda,\gamma,\rho} w', \nu : ((\mathcal{L} \cup \mathcal{I}) \to \mathbb{U})) \mapsto (\mu_Q(w, \nu \restriction \mathcal{L}) \xrightarrow{\mu_L(\lambda,\nu\restriction\mathcal{I})} \mu_Q(w', \nu \text{ } \mathit{after} \text{ } \rho))
\end{aligned}$$

When the number of possible valuations for $\mathcal{L}$ and $\mathcal{I}$ is finite, the transformation is always possible to an LTS with finite number of states.

An example of this transformation is shown in Figure 2.5. The label 'do(1)' in the LTS is a textual representation of the gate 'do' plus a valuation of the interaction variable 'd'. The text on the nodes indicate from which location and valuation the state was created. The node labelled '$w_0, N = 2$' is an example of an unreachable state.



(a) The STS

(b) The LTS

Figure 2.5: An example of a transformation of an STS to an LTS

2.3.5 Coverage

Coverage metrics do not only apply to LTSs, they can also be used on STSs. The simplest metric to describe the coverage of an STS is the location and switch-relation coverage, which express the percentage of locations and switch relations tested. Measuring state and transition coverage of an STS is possible using the LTS from the STS to LTS mapping.

2.4 Graph Grammars

A *Graph Grammar* (GG) is also a specification of system behavior like LTSs and STSs are. A GG is composed of a set of graph transformation rules. These rules indicate how a graph can be transformed to a new graph. These graphs are called *host graphs*. The rules are composed of graphs themselves, which are called *rule graphs*.

The rest of this section is ordered as follows: first, graphs, host graphs, rule graphs and graph transformation rules are explained. Then, the definition of a *Graph Transition System* (GTS) is given. An example of a GG and a GTS is then given. Finally, the definition of IOGGs is given. For a more detailed overview of GGs, we refer to [22, 9, 1].

Definition 2.4.1. Graphs
A *graph* is composed of nodes and edges. In this report, we assume a universe of nodes $\mathbb{V} = \mathbb{W} \uplus \mathbb{U} \uplus \mathcal{V} \uplus 2^{\mathcal{T}}$, where $\mathbb{W}$ is the universe of standard graph nodes. $\mathbb{E}$ is the universe of edges between two nodes in $\mathbb{V}$.

Definition 2.4.2. Host graphs
A host graph G is a tuple $\langle V_G, E_G \rangle$, where:

- $V_G \subseteq (\mathbb{W} \uplus \mathbb{U})$ is the node set of G
- $E_G \subseteq (V_G \backslash \mathbb{U} \times L \times V_G)$ is the edge set of G

Figure 2.6 shows an example of a host graph. Here, $n_1, n_2 \in \mathbb{W}$ are the *identities* of the nodes. The other four nodes are values in $\mathbb{U}$.

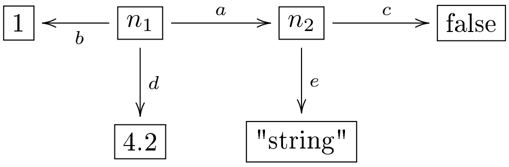


Figure 2.6: An example of a host graph

Definition 2.4.3. Rule graphs
A rule graph H is a tuple $\langle V_H, E_H \rangle$, where:

- $V_H \subseteq (\mathbb{V} \backslash \mathbb{U})$ is the node set of H
- $E_H \subseteq (V_H \times L \times V_H)$ is the edge set of H

In addition, the following must hold:

- $\forall z \in V_H \cap 2^{\mathcal{T}} \,.\, var(z) \subseteq V_H$ - The variables used in the terms must be present as nodes in the rule graph.
- $\forall z \in V_H \cap z \in \mathcal{V} \,.\, \exists(\_, \_, z) \in E_H$ - If a variable is used in a rule graph, it needs context. Therefore, there must be an edge with the variable node as target.

Figure 2.7 shows an example of a rule graph. Here, $r_1, r_2 \in \mathbb{W}$ are the node identities, $x_1, x_2 \in \mathcal{V}^{int}$ and $\{x_1 + 1, x_2\} \in 2^{\mathcal{T}}$. The set of terms is mapped as a node to the same value. This mapping is explained in the next definition. The consequence is that this node implicitly expresses the relation $x_1 + 1 = x_2$.

Definition 2.4.4. Morphisms
A graph g has a *morphism* to a graph g' if there is a structure-preserving mapping from the nodes and the edges of g to the nodes and the edges of g' respectively. A graph g has a partial morphism to a graph g' if there are elements in g without an image in g'.

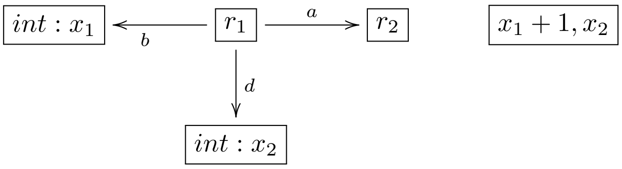


Figure 2.7: An example of a rule graph

Definition 2.4.5. Images
Given a morphism from a graph g to a graph g', a node or edge z in g has an *image* z' in g'. z is then a *pre-image* of z'. For each type of node, an explanation is given:

- A variable $v \in \mathcal{V}^s, s \in S$, has an image i in a host graph if $i \in \mathbb{U}^s$.
- A node $z \in 2^{\mathcal{T}}$ has an image i in a host graph if i is the valuation of all terms in z.

Definition 2.4.6. Transformation rules
A transformation rule r is a tuple $\langle LHS, NAC, RHS, l\rangle$, where:

- *LHS* is a rule graph representing the left-hand side of the rule
- *NAC* is a set of rule graphs representing the negative application conditions
- *RHS* is a rule graph representing the right-hand side of the rule
- $l \in L$ is the label of the rule

There exist implicit partial morphisms from the *LHS* to each rule graph in *NAC* and from the *LHS* to the *RHS* by means of the node identities. These morphisms are *rule graph morphisms.*

Definition 2.4.7. Creators and erasers
A *creator* is an edge or node in the *RHS* of a rule, that is not in the *LHS* of the rule. An *eraser* is an edge or node in the *LHS* of a rule that is not in the *RHS* or a rule.

Definition 2.4.8. Rule matches
A rule r has a *rule match* on a host graph G if its *LHS* has a morphism to G and none of the graphs in *NAC* have a morphism to G. The morphism of the *LHS* to a host graph is a *match morphism.*

Definition 2.4.9. Rule transitions
After the rule match is found, all nodes and edges in *LHS* that do not have an image in *RHS*, are removed from G. All elements in *RHS* that do not have a pre-image in *LHS*, are added to G. The result of this process is called a *rule transition*, denoted by: $G \xrightarrow{r,m} G'$, where $m \in M$ is the morphism of the *LHS* to G.

Figure 2.8 shows an example of the initial graph G_0, one rule of a GG and the corresponding rule match. G_0 can be represented by $\langle\{n1, n2\}, \{\langle n1, a, n1\rangle, \langle n1, A, n2\rangle, \langle n2, B, n2\rangle\}\rangle$. The *LHS* of the rule has a match in G_0. Neither *NAC1* and *NAC2* have a match in G_0, because the edge with label C does not exist in G_0. The new graph after applying the rule is G_1.

Definition 2.4.10. Graph Grammars
A graph grammar is a tuple $\langle R, G_0\rangle$, where:

- R is a set of graph transformation rules
- G_0 is the initial graph

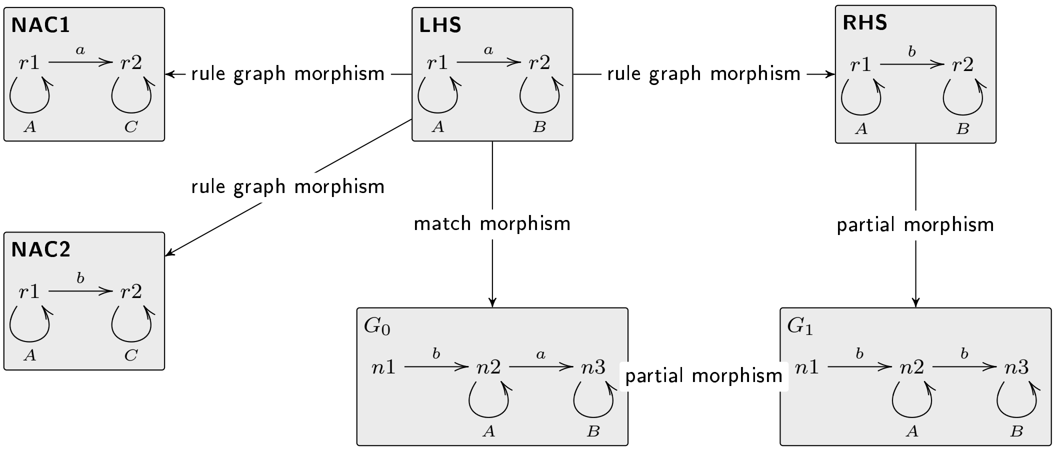


Figure 2.8: An example of a graph transformation

By repeatedly applying graph transformation rules to the start graph and all its consecutive graphs, a GG can be explored to reveal a *Graph Transition System* (GTS). This transition system consists of graphs connected by rule transitions.

Definition 2.4.11. Graph Transition Systems
A graph transition system is a tuple $\langle \mathcal{G}, R, M, U, G_0 \rangle$, where:

- $\mathcal{G}$ is a set of graphs
- $L \subseteq R \times M$ is a set of labels
- $U \subseteq \mathcal{G} \times L \times \mathcal{G}$ is the rule transition relation
- $G_0 \in \mathcal{G}$ is the initial graph

Let $K = \langle R, G_0 \rangle$. A GTS $O = \langle \mathcal{G}, R, M, U, G_0 \rangle$ is derived from K by the following. $\mathcal{G}, M, U$ are the smallest sets, such that:

- $G_0 \in \mathcal{G}$
- if $G \in \mathcal{G}$ and $G \xrightarrow{r,m} G'$ then $G' \in \mathcal{G}, (r, m) \in L, (G \xrightarrow{r,m} G') \in U$

In order to specify stimuli and responses with GGs, a definition is given for an *Input-Output GG* (IOGG).

Definition 2.4.12. Input-Output Graph Grammars
An IOGG is a tuple $\langle R_Y, G_0 \rangle$, where $R_Y \subseteq R \times Y$ are input-output transformation rules.

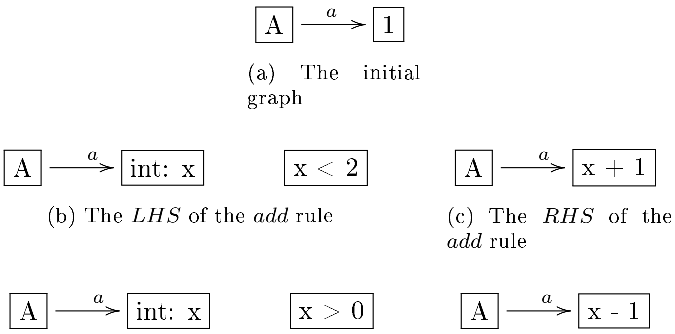
Exploring an IOGG leads to an *Input-Output Graph Transition System* (IOGTS).

Definition 2.4.13. Input-Output Graph Transition Systems
An IOGTS is a tuple $\langle \mathcal{G}, R_Y, M, U_Y, G_0 \rangle$, where:

- $L_Y \subseteq R_Y \times M$ are the input-output labels
- $U_Y \subseteq \mathcal{G} \times L_Y \times \mathcal{G}$ are the input-output rule transitions

Definition 2.4.14. Rule priorities
A graph grammar with *rule priorities* P assigns a priority $p \in \mathbb{N}$ to a $r \in R$, such that $r \mapsto p \in P$. The definition of GTSs is extended with this definition of rule priorities and the following:
$r_1, r_2 \in R, G \in \mathcal{G}.(G \xrightarrow{r_1,m_1} G') \in U \wedge P(r_1) > P(r_2) \rightarrow (G \xrightarrow{r_2,m_2} G') \notin U$

In the graph grammar in Figure 2.9, the 'add' rule produces a rule transition to a graph, where the *sub* rule produces a rule transition back to the start graph. Suppose $P(add) > P(sub)$, then the 'sub' rule does not have a outgoing rule transition from the start graph.



(a) The initial graph

(b) The *LHS* of the *add* rule

(c) The *RHS* of the *add* rule

(d) The *LHS* of the *sub* rule

(e) The *RHS* of the *sub* rule

Figure 2.9: Priority rules

2.5 Tooling

2.5.1 ATM

ATM is a model-based testing web application, developed in the Ruby on Rails framework. It has been used to test the software of several big companies in the Netherlands since 2006. It is under continuous development by Axini.

The UML sequence diagram for ATM is shown in Figure 2.10.

The tool functions as follows:

1. An STS is given to an STS Engine, which keeps track of the current location and variables. The user starts the test and gives a 'depth', indicating how many stimuli should be tested. The variable i stands for the current iteration of *loop* and is initially set to 0. The variable $verdict$ is initially set to $'pass'$.
2. The STS Engine gives the possible switch relations from the current location to the Test Manager. It chooses an enabled switch relation based on a test strategy, which can be a random strategy or a strategy designed to obtain a high location/switch relation coverage. The valuation of the variables in the guard are also chosen by a test strategy, which can be a random strategy or a strategy using boundary-value analysis. The choice is represented by an instantiated switch relation.
3. The instantiated switch relation is given to the Test Execution component as an *abstract stimulus*. The term abstract indicates that the instantiated switch relation is an abstract representation of some computation steps taken in the SUT. For instance, a transition with label '?connect' is an abstract stimulus of the actual setup of a TCP connection between two distributed components of the SUT.
4. The translation of an abstract stimulus to a concrete stimulus is done by the Adapter. This component has to be programmed by the tester such that the abstract stimulus is correctly translated to a concrete stimulus. This component provides the stimulus to the

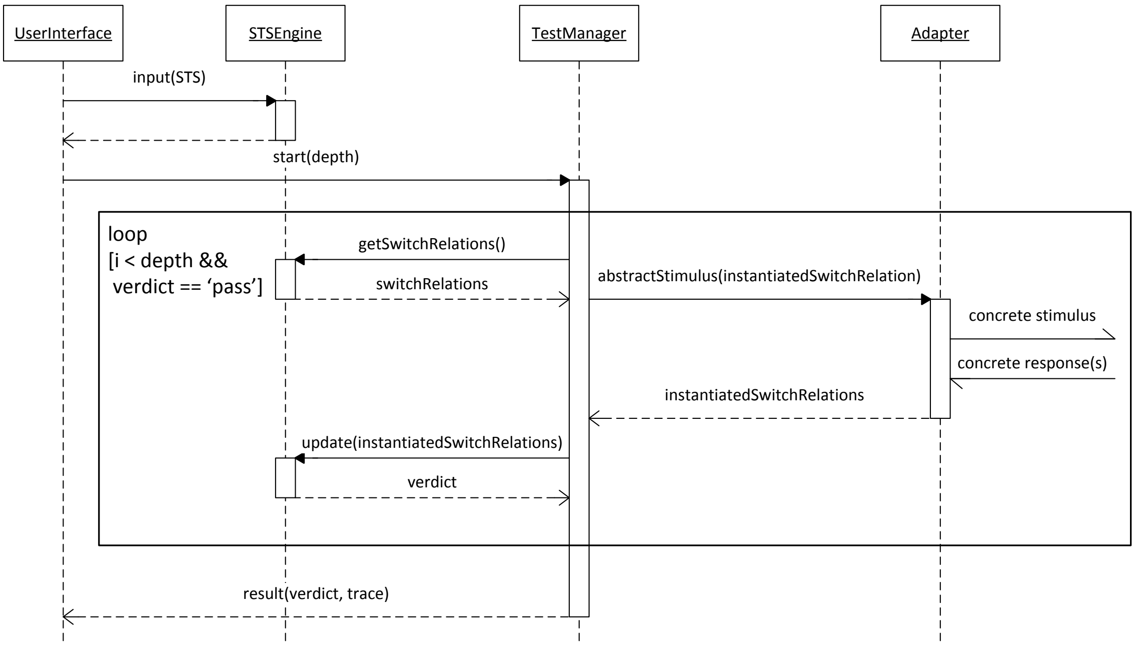


Figure 2.10: ATM sequence diagram

SUT. When the SUT responds, the Adapter translates this response to an abstract response. For instance, the Adapter receives an HTTP response that the TCP connect was succesful. This is a concrete response, which the Adapter translates to an abstract response, such as an instantiated switch relation with gate '!ok'. The SUT can also give multiple responses. As with the stimuli, the tester has to program the translation from concrete responses to abstract responses. The Test Manager is notified with these abstract responses.

5. The Test Manager updates the STS engine with the chosen abstract stimuli and received abstract responses. If this is possible according to the STS, a pass verdict is given, otherwise a fail verdict is given. The Test Manager updates the *verdict* variable accordingly. The loop continues as long as all responses are according to the specification and the required number of tested stimuli has not been reached. The test is stopped at a fail verdict, because the SUT has entered an invalid state and the STS engine cannot give possible switch relations any more. For instance, an error could have occurred, which is an invalid response and makes continuing impossible.

6. When the Test Manager finishes it gives a pass verdict for this test if all verdicts given by the STS engine were a pass verdict. Otherwise, the result is a fail verdict. Also a trace is given of all chosen and observed instantiated switch relations. This can be used to calculate coverage information for the test and to allow the SUT or the STS to be fixed in case of a fail verdict.

2.5.2 GROOVE

GROOVE is an open source, graph-based modelling tool in development at the University of Twente since 2004 [23]. It has been applied to several case studies, such as model transformations and security and leader election protocols [6]. The UML sequence diagram for GROOVE is shown in Figure 2.11.

The tool functions as follows:

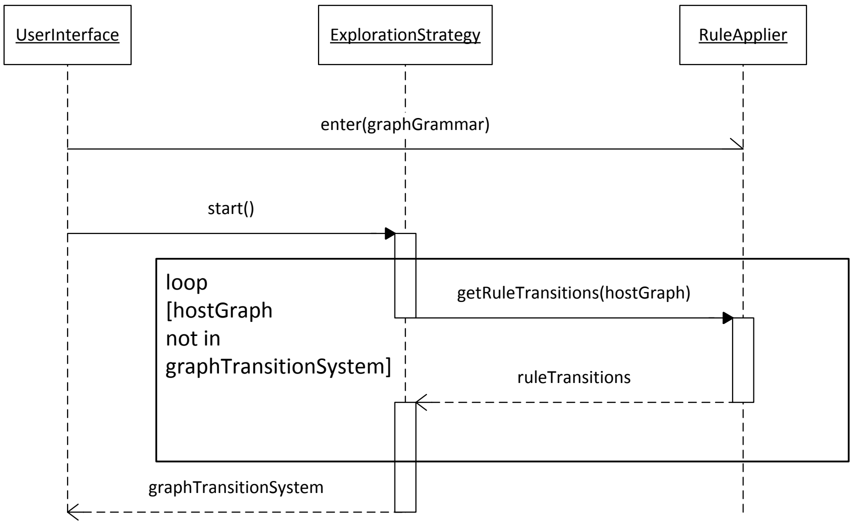


Figure 2.11: GROOVE sequence diagram

1. A graph grammar is given as input to a RuleApplier component, which determines the possible rule transitions.
2. The user starts an ExplorationStrategy. This strategy explores all possible host graphs and rule transitions. The possible rule transitions from the initial graph state are obtained from the RuleApplier and a rule transition is chosen, based on the exploration strategy. The target host graph of the chosen rule transition is again given to the RuleApplier until no more new host graphs can be explored.
3. The ExplorationStrategy returns the explored GTS to the UserInterface.

2.5.3 GROOVE visual elements

Labels and flags Nodes in GROOVE have several kinds of labels: regular labels, type labels and flags. Figure 2.12 shows a node with a type label (bold), two flags (italic) and two regular labels. Nodes in GROOVE can have one type, indicated by the type label. Typing is not explained further in this report[1]. A node can have multiple regular labels and flags.

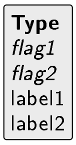


Figure 2.12: GROOVE labels and flags

Rule node matching Nodes in a rule graph in GROOVE can also match each other, by connecting them with an equals ‘=’ labelled edge. This means that any images of both nodes have to be the same. Figure 2.13a shows an example of this. Nodes in a rule graph in GROOVE can also explicitly not match each other, by connecting them with an not-equals ‘!=’ labelled edge. This means that any images of both nodes cannot be the same. Figure 2.13b shows an example of this.

[1] See the documentation of GROOVE for more information: http://groove.cs.utwente.nl/doc/

node1
=
node2
node1
!=
node2

(a) Matching rule nodes

(b) Non-matching rule nodes

Figure 2.13: Node matching in GROOVE rule graphs

Colors Rule graphs in GROOVE combine *LHS*, *RHS* and *NAC* into one rule graph. The colors on the nodes and edges in GROOVE rules represent whether they belong to the *LHS*, *RHS* or *NAC* of the rule. See Figure 2.14 for an example.

1. normal line (black): This node or edge is part of both the *LHS* and *RHS*.
2. dotted line (red): This node or edge is part of the *NAC* only.
3. thick line (green): This node or edge is part of the *RHS* only.
4. dashed line (blue): This node or edge is part of the *LHS* only.

dotted line is niet helemaal correct beter woord?

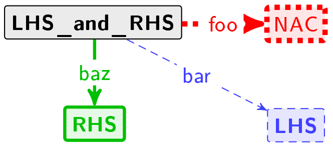


Figure 2.14: GROOVE rule graph colors

Variable nodes and terms Variable nodes in GROOVE are represented by their type: ‘int’, ‘bool’, ‘real’ and ‘string’ for integer, boolean, real and string variables respectively. Figure 2.15 shows two integer variable nodes and the constant integer node ‘1’. The diamond shaped node is a term node. It has two *argument* edges π_0, π_1 and a *result* edge ‘int:add’. The term represented here is the addition of two integers, the first one being an integer variable, the second being the number 1. When this rule matches a host graph, the target variable node of the result edge is set to the result of the term; in this case the image of the first variable node plus one.

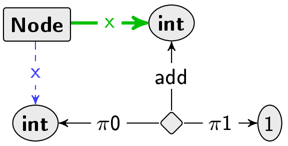
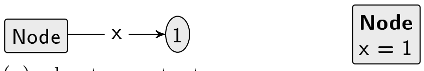


Figure 2.15: Terms in GROOVE rule graphs

Term shorthand notation A rule node with an edge to a constant is shortened to a label on the node. Figure 2.16a shows a node with an edge labelled ‘x’ to the constant ‘1’. Figure 2.16b shows the shorthand notation of this edge as the label ‘x = 1’ on the source node of the edge. Terms can also be shortened. The rule graph in Figure 2.15 can be shortened to the rule graph in Figure 2.17.

Rule transition parameters Rule transitions can have parameters in GROOVE. Figure 2.18a shows a rule where the node variables have a number in the top right of the node. These numbers indicate that the value of the variables are placed as parameters on the rule transition, in the order indicated by the numbers. This rule matches the host graph in Figure 2.18b. The result of applying the rule twice is the GTS in Figure 2.18c.



(a) edge to constant

(b) shorthand notation

Figure 2.16: Constant shorthand notation in GROOVE

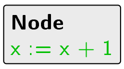


Figure 2.17: Term shorthand notation in GROOVE

Quantification GROOVE supports quantification operations over nodes in rule graphs. Figure 2.19 shows a simple example. The 'forall' operator here matches all nodes typed 'Node'. GROOVE also supports the 'exists' operator and nesting of operators, however this is out of scope for this report. The 'forall' operator will be used in the model examples to perform operations over sets of nodes, such as in this rule: all self-edges labelled 'x' on nodes typed 'Node' are deleted from the host graph.

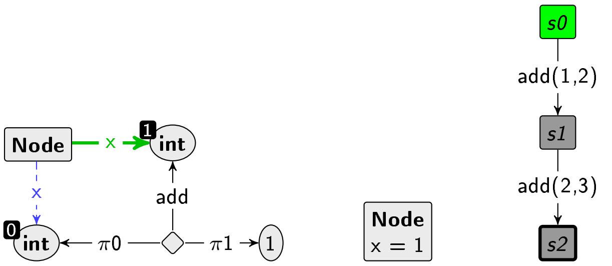
2.5.4 Example GROOVE graph grammar

The running example from Figure 2.4 is displayed as a graph grammar, as visualized in GROOVE, in Figure 2.20. The *LHS*, *RHS* and *NAC* of a rule in GROOVE are visualized together in one graph. Figures 2.20b, 2.20c and 2.20d show three rules. Figure 2.20a shows the start graph of the system.

The rules can be described as follows:

1. 2.20b: 'if a player has the turn and he has not thrown the die yet, he may do so.'
2. 2.20c: 'if a player has the turn and he has thrown the die and this number is larger than zero, he may move one place and then it is as if he has thrown one less.'
3. 2.20d: 'if a player has finished moving (number thrown is zero), the next player receives the turn.'

The graph is transformed after the rule is applied. The resulting graph after the transformation is the new state of the system and the rule is the transition from the old state (the graph as it was before the rule was applied) to the new state. Figure 2.21 shows the IOGTS of one *?throws* rule application on the start graph. Note that the *?throws* is an input, as indicated by the '?'. State s_1 is a representation of the graph in Figure 2.20a. Figure 2.22 shows the graph represented by s_2.



(a) Rule graph (b) Host graph (c) GTS

Figure 2.18: Rule transition parameters in GROOVE

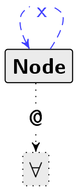
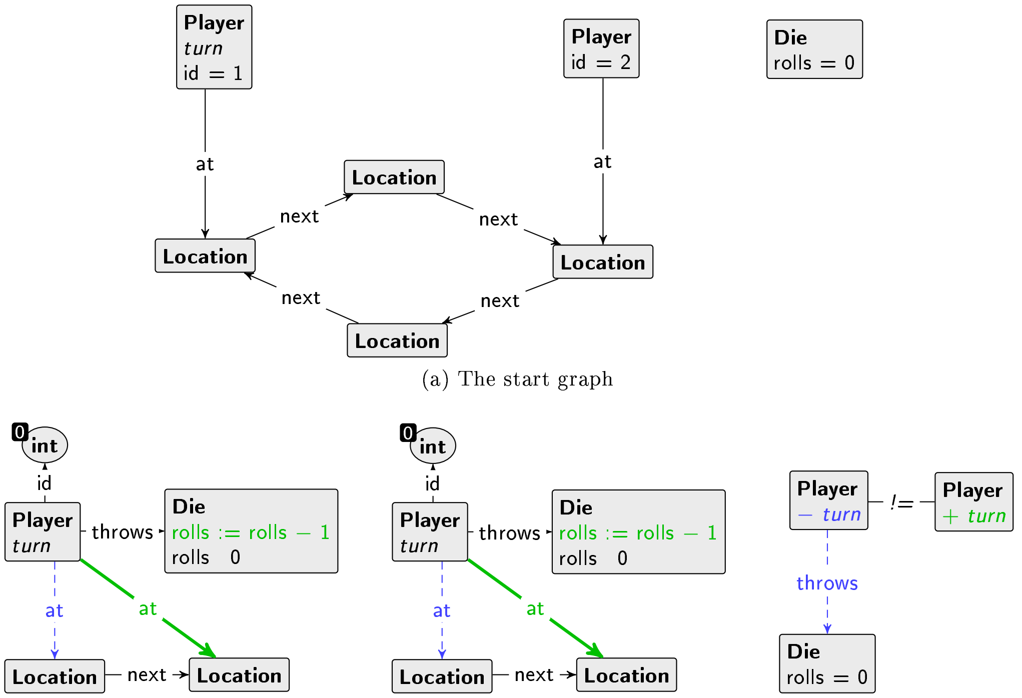


Figure 2.19: An example of quantification in GROOVE



(a) The start graph

(b) The throw rule (c) The move rule (d) The next turn rule

Figure 2.20: The graph grammar of the board game example in Figure 2.4

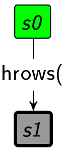


Figure 2.21: The GTS after one rule application on the board game example in Figure 2.20

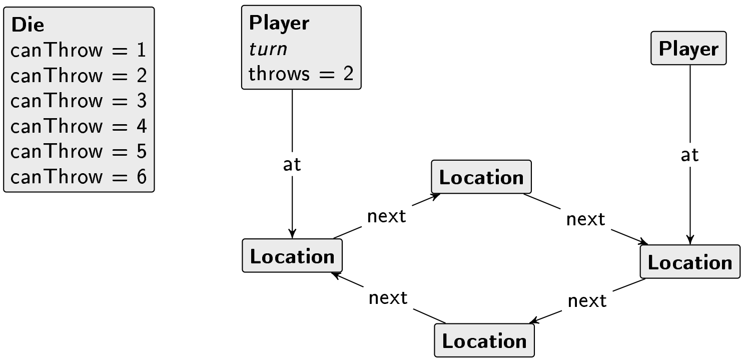


Figure 2.22: The graph of state $s2$ in Figure 2.21

Chapter 3

From Graph Grammar to STS

3.1 Requirements considerations

From the research goals we derive three high level requirements:

1. The system should be able to perform model-based testing on GGs
2. The system should be validated so it should allow coverage metrics to be measured, because as seen in the previous chapter, coverage statistics are a useful and objective metric to describe how much of the model is tested
3. The system should be validated using case studies so it should be efficient enough to handle large models

IOGGs In order to fulfill the first requirement, stimuli and responses have to be obtained from a GG. ATM uses an IOSTS, where the instantiated switch relations represent a stimulus to or a response from the SUT. GGs have no notion of inputs and outputs, therefore IOGGs have to be used as the model formalism. IOGGs can be explored to IOGTSs and the I/O labels of the IOGTS can be used to represent abstract stimuli/responses.

On-the-fly vs. offline exploration The exploration of a GG can be done in two ways: *on the fly*, rule transitions are explored only when chosen by ATM, or *offline*, the GG is first completely explored and then sent to ATM. On-the-fly model exploration works well on large and even infinite models. However, coverage statistics cannot be calculated with this technique. The number of states (graphs) and rule transitions the model has when completely explored are not known, so a percentage cannot be derived. As coverage statistics are an important metric, the offline model exploration is chosen for GRATiS.

Data values An IOGTS can potentially be infinitely large, due to the range of data values. This is a potential risk for the validation of the system. A model that is more efficient with data values is an STS. The setup of GRATiS is therefore to transform the IOGG directly to an IOSTS. This transformation should be done efficiently to fulfill the third requirement. Note that the second requirement is still met, because location and switch relation coverage can be calculated on the IOSTS.

Taking these requirements into account, the method to achieve the goal of model-based testing on GGs is the following three steps:

1. Create an IOGG by assigning I/O types to the graph transformation rules of a GG
2. Create an IOSTS from the IOGG
3. Perform the model-based testing on the IOSTS

The rest of this chapter describes a declaritive definition for creating an IOSTS from an IOGG.

3.2 From IOGG to IOSTS

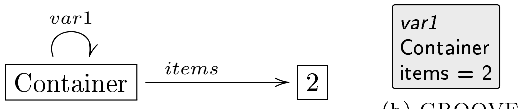
3.2.1 Variables in GGs

The location variables in an STS represent an aspect of the modelled system. For instance, if a system keeps track of the number of items in containers, the STS modelling this system could have integer location variables $items_1..items_n$. GGs do not have this kind of variables. The variable nodes in rule graphs are used to match a value in a host graph, which is only available to that rule. A definition of persistent variables in GGs is needed in order to define location variables from an IOGG.

Definition 3.2.1. GG variables
A GG variable is a tuple $\langle l_a, l_v \rangle$, where:

- l_a is the *variable label*, which is part of an edge (z_a, l_a, z_a), where z_a is called the *variable anchor*
- l_v is the *value label*, which is part of a *value edge*: (z_a, l_v, z), with $z \in \mathbb{U}$.

The item/container example modelled in a graph grammar could be a graph node representing a container with an edge labelled 'items' to an integer node. This is shown in Figure 3.1a. The variable $(var1, items)$ is now represented by this graph. Figure 3.1b shows the same example in GROOVE. The variable edge is omitted here and the variable label is represented by the flag *var1*. This convention for GROOVE is used in the rest of this report.



(a) Formal

(b) GROOVE

Figure 3.1: Example of a GG variable

3.2.2 Graph exploration with point algebra

The IOGG cannot be directly made into an IOSTS, without using the IOGTS. To avoid the problem of a potentially infinitely large IOGTS, the point algebra is used. Using the point algebra when exploring a GG has two effects:

1. The host graphs that differ only in value nodes are collapsed into one
2. The variable nodes in rule graphs can have at most one image in the host graph

The actual values of the value nodes in host graphs and of the variable nodes in rule graphs are not needed to make the IOSTS, which is explained in the next section. Using the point algebra for the exploration poses some constraints, which are explained in section 3.4.

3.2.3 The IOGG to IOSTS definition

First the declaritive definition is given, then each part of the definition is described in more detail.

Definition 3.2.2. IOGG to IOSTS
Let K be an IOGG. From K we define an IOSTS J. The first step is to explore K using the point algebra $\mathcal{P}$ to an IOGTS $O_{\mathcal{P}}$. The elements of J can be obtained as follows:

- $W = \mathcal{G}$
- $w_0 = G_0$
- $\mathcal{L} = \mathcal{V}_{GG}$
- $\imath = \phi_\imath$, where $\phi_\imath$ is defined below
- $\Lambda = R$
- $\mathcal{I} = \mathcal{V}$
- $D = U$, where γ, ρ are given by functions defined below

Locations The locations abstract from data values, exactly like host graphs do under the point algebra. Therefore, the set of locations in J are the set of host graphs in $O_{\mathcal{P}}$. The initial location in J is also equal to the start graph of $O_{\mathcal{P}}$.

Location variables GG variables were defined to have location variables in GGs. Therefore, it is no surprise that the set of location variables in J is exactly the set of GG variables in $O_{\mathcal{P}}$. This poses some constraints on creators/erasers for the variable anchors, variable edges and value edges. This is explained in detail in section 3.4.

Initialization function The initial value of the location variables is defined by the start graph. The start graph contains the GG variables and their initial values. This poses a constraint on the start graph, which is explained in section 3.4.

Definition 3.2.3. Initialization function
$\phi_\imath : \ (l_a, l_v) \mapsto z \mid (z_a, l_a, z_a) \in E_{G_0} \wedge (z_a, l_v, z) \in E_{G_0}$

Gates The gate of a switch relation represents the stimulus to or response from the SUT. In an IOGG, the rules represent the stimuli and responses. Therefore, the set of gates Λ is equal to the set of rules R.

Interaction variables Interaction variables are used by the gates to represent a stimulus or response variable. The variable nodes in rule graphs are this representation. The set of interaction variables $\mathcal{I}$ is equal to the set of variable nodes $\mathcal{V}$. The arity of a rule is defined by the variable nodes in the *LHS* of the rule:

$arity(r) = |\mathcal{V} \cap V_{LHS}|$

Switch relations A rule transition $G \xrightarrow{r,m} G' \in U$ is mapped to a switch relation $(G \xrightarrow{r,\phi_\gamma,\phi_\rho} G') \in D$. Note that the set of locations is the set of host graphs, therefore G and G' represent the source and target location of the switch relation. Also, the set of rules is chosen to be the set of gates, therefore r represents the gate of the switch relation. The function ϕ_γ obtains the guard as a term and ϕ_ρ replaces the ρ function. These functions are defined in the next paragraphs.

Guard The guard of a switch relation restricts the use of the switch relation based on the values of the variables. In a GG, a rule is restricted by the terms. The variables used in the terms are interaction variables. Therefore, the first part of the guard is constructed by joining the terms for each term node.

Definition 3.2.4. First guard function
$\phi 1_\gamma : \bigwedge_{z \in V_{LHS} \cap 2^{\mathcal{T}},\, t_1 \in z,\, t_2 \in z} t_1 = t_2$

We apply this to the rule graph in Figure 3.2. This rule graph contains one term node, which in turn contains two terms. Formally, the first part of the guard for this rule graph is:

$(x_1 + 1 = x_1 + 1) \wedge (x_1 + 1 = x_2) \wedge (x_2 = x_2) \wedge (x_2 > 3 = x_2 > 3) \wedge (x_2 > 3 = \mathit{true}) \wedge (\mathit{true} = \mathit{true})$

When $t_1 = t_2$, the equation for this part is not useful, therefore it is omitted from now on.

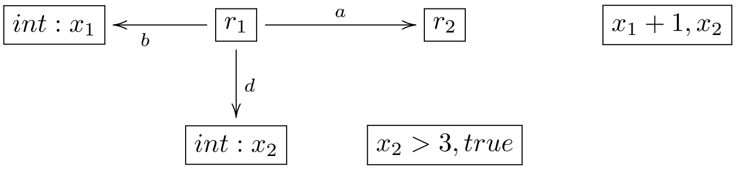


Figure 3.2: a rule graph

This first part of the guard contains only the interaction variables. In an STS, the values for x_1 and x_2 can be chosen such that the guard holds. However, the variable nodes cannot have just any value; their value is determined by their image in the host graph. This image can be the value of a GG variable. Therefore we add a second part to the guard to link the interaction and location variables.

Definition 3.2.5. Second guard function
$\phi 2_\gamma : \bigwedge_{(l_a,l_v) \in \mathcal{V}_{GG},\, z \in \mathcal{V} \cap V_{LHS} \mid (z_a,l_a,z_a) \in E_G \wedge (z_a,l_v,m(z)) \in E_G} (l_a, l_v) = z$

We apply this to the rule graph in Figure 3.2 and the host graph in Figure 3.3. The rule has a match in the host graph. The second part of the guard for this rule is:

$(\mathit{var}1, b) = x_1 \wedge (\mathit{var}1, d) = x_2$

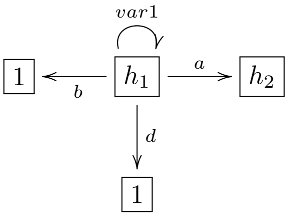


Figure 3.3: a host graph

Definition 3.2.6. Guard function
The complete guard is created by:

$\phi_\gamma : \phi 1_\gamma \wedge \phi 2_\gamma$

Update mapping When a value edge is erased from a graph and a new value edge is created with the same label and variable anchor, this inidicates an update for the corresponding GG variable. The update mapping for the switch relation represented by this rule transition should map the matched GG variable to the interaction variable represented by the target of the new value edge.

Definition 3.2.7. Update mapping function
$\phi_\rho : (l_a, l_v) \mapsto z' \mid (z_a, l_a, z_a) \in E_G \wedge (z, l_v, z') \in E_{RHS} \wedge m(z) = z_a$

This poses some constraints on the eraser/creator edges for value edges, which are explained in section 3.4. These constraints ensure that ϕ_ρ is bijective.

3.3 Rule priorities

Rule priorities are not taken into account in the definition above. First, we show why the definition above does not work with rule priorities. Then we show a method of including rule priorities in the IOSTS.

We apply the definition to the GG in Figure 2.9. We assume the rules both are stimuli, i.e. each is a I/O transformation rule (r, *input*), where r is the respective rule. All host graphs explored by this GG are isomorphic under the point algebra, so they represent the same location. The IOSTS obtained from this IOGG is in Figure 3.4, with $\imath = \{x \mapsto 25\}$. This IOSTS is wrong, because the '?sub' switch relation can be taken from the start, whereas the '?sub' rule in the IOGG cannot be applied to the start graph, because it has a lower priority than the '?add' rule.

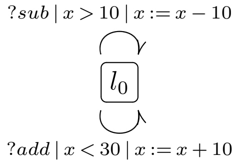

$?sub \mid x > 10 \mid x := x - 10$
l_0
$?add \mid x < 30 \mid x := x + 10$

Figure 3.4: A wrong STS transformation of the graph grammar in Figure 2.9

The solution is shown in Figure 3.5. The negated guard of the '?add' switch relation is added to the '?sub' switch relation. The optimized guard for this switch relation is 'x >= 30' of course, but this shows the main principle: for a rule transition u and a set of rule transitions U with higher priority rules and the same source graph as u, the negated guard of the switch relations represented by U must be added to the guard of the switch relation represented by u. In the example, the 'x < 30' guard is negated to '!(x < 30)' and added to yield the 'x > 10 && !(x < 30)' guard.

This problem only arises when the switch relation represented by the rule transition has a guard; if the rule has no constraints on data and if it is applicable on one graph state, it is applicable on all isomorphic graph states under the point algebra. Therefore, rule transitions with lower priority rules do not exist from that graph state and the respective switch relations also do not exist.

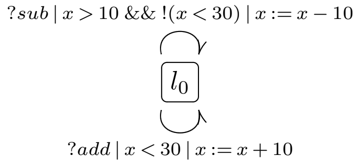

?sub | x > 10 && !(x < 30) | x := x − 10
l0
?add | x < 30 | x := x + 10

Figure 3.5: A correct STS transformation of the graph grammar in Figure 2.9

3.4 Constraints

This section describes the constraints on the definition in section 3.2.

GG variable persistency All location variables in an STS are initialized to some value and no new variables are added. In a GG, it is possible to erase and create new variables in the transformation rules. Therefore, we pose the following constraints:

- All GG variables must be present in the start graph
- Variable anchors and variable edges cannot be erased or created
- A transformation rule may erase a value edge if and only if it creates a new edge with the same source node and label as the erased value edge. The target node of the new edge must be a value node of the same sort as the target node of the erased edge

Unique GG variables The GG variables to location variables mapping has to be bijective. The previous constraints ensure the GG variables are always present, however they do not ensure their uniqueness. Therefore, we pose the following constraints:

- The variable labels have to be unique; there cannot be two variable edges with the same label in the start graph
- The value labels for the value edges with the same source node have to be unique; there cannot be two edges with the same variable anchor with the same label in the start graph

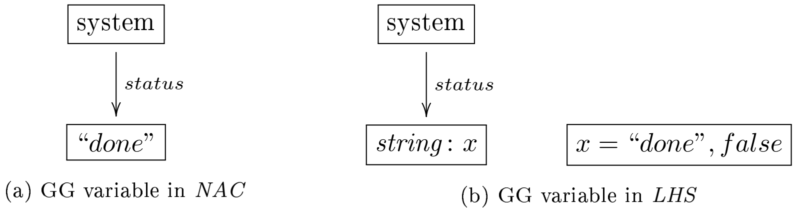
No GG variables in NACs In graph transformation rules, it is common to create restrictions on data using a NAC. For example, the NAC in Figure 3.6a expresses that the rule cannot be applied when the system is in state 'done'. When using the point algebra, a problem occurs when a value edge is part of a NAC. The rule containing such a NAC will never match any host graph, because:

(a) the value edge is present in every host graph as stated in the previous constraints

(b) the target value node of the value edge will always be an image of the target variable node of the pre-image of the value edge. This is assuming the variable node in the NAC is of the same sort as the value node, otherwise the NAC would never have a match, because of the constraint on the sort of the value node of a value edge. Such a NAC would add nothing to the rule.

Therefore we pose the following constraint: A NAC may never match the value edge and value node of a GG variable.

The correct way of modelling the example when using the point algebra is shown in Figure 3.6b. The term node $\{x = \text{“}done\text{”}, false\}$ is in the LHS of the rule. Under the point algebra, both terms evaluate to the same boolean value. Therefore, an image for this term node can always be found, but the guard of the switch relation will be $(x = \text{“}done\text{”}) = false$. This leads to some cases



(a) GG variable in *NAC*

(b) GG variable in *LHS*

Figure 3.6: Node creating rule without structural constraint

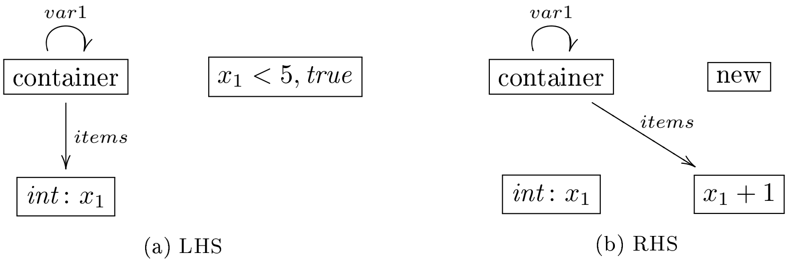
where the guard of the switch relation can never be satisfied. The following paragraph describes such a case.

Data constraints in node creating rules A case where a guard can no longer be satisfied is shown in Figure 3.7. This figure shows the *LHS* and *RHS* of a rule in the container-items example. The rule adds an item to the container unless it is full, i.e. has five items. If an item is added, a new node labelled 'new' is created in the host graph. Under the point algebra, this rule always matches the host graph and thus this rule creates an infinite number of nodes. Therefore, the exploration never ends. This means that an IOGG with such a rule has an infinite number of graphs and the corresponding IOSTS has an infinite number of switch relations. However, the guards of the switch relations can no longer be satisfied when the location variable corresponding to the GG variable (*var1*, *items*) reaches 5.

The rules in an IOGG which create new nodes or edges and have constraints on data should either:

- erase the same amount of nodes and edges as it creates
- have a constraint on the created node or edge, e.g. the node to create is part of the *NAC* of the rule

The first constraint guarantees the exploration halts. The second constraint is only a guarantee if the node is the only element in the *NAC*, otherwise it is possible that the *NAC* does not match the graph even with the node in it.



(a) LHS

(b) RHS

Figure 3.7: Node creating rule with data constraint

Chapter 4

Implementation

4.1 General setup

GROOVE has several exploration strategies for exploring a GG. The SymbolicStrategy is added as exploration strategy. It contains functionality to build an IOSTS from an IOGG. Figure 4.1 shows the UML sequence diagram of GRATiS. The functionality of GROOVE and ATM as described in Figures 2.10 and 2.11 is omitted where possible.

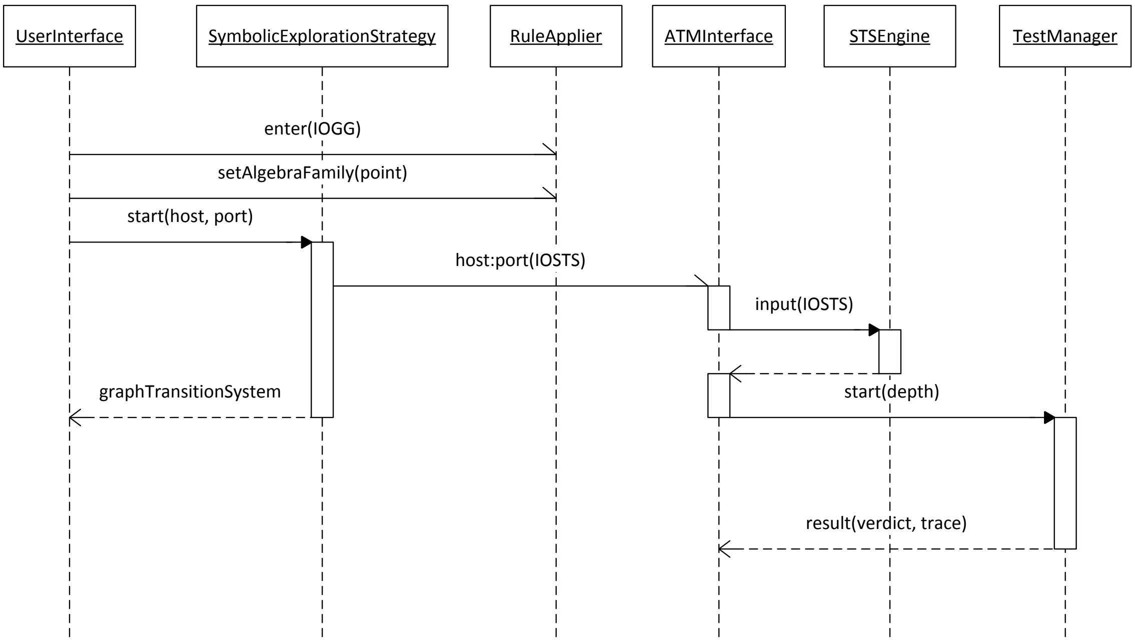


Figure 4.1: GRATiS sequence diagram

1. The user enters an IOGG in the RuleApplier.
2. The user sets the algebra family of the IOGG to to the point algebra.
3. The user starts the SymbolicExplorationStrategy with a host and port. The communication between the RuleApplier and the strategy is omitted here. However, it should be noted that the point algebra is used by the RuleApplier here to create the rule transitions.

4. The SymbolicExplorationStrategy creates an IOSTS from the IOGTS and sends the IOSTS to the given host and port.
5. The IOSTS is given to the STSEngine. The ATMInterface takes the role of the UserInterface in ATM.
6. The ATMInterface starts the testing using a configurable depth parameter. The communication between the TestManager, STSEngine and Adapter is omitted here.
7. The TestManager returns the test results. The test results are stored in a database and are viewable through the user interface of ATM.

4.2 Description of added functionality

This section covers in detail the added functionality to GROOVE and ATM.

4.2.1 GROOVE exploration strategy

Figure 4.2 shows the class diagram of the added exploration strategy interface. The Symbolic-ExplorationStrategy has an exploration strategy such as the BreadthFirstStrategy to explore the GTS. The remote exploration strategy extends the symbolic exploration strategy.

The user starts the remote exploration strategy with a host and port. This strategy starts a breadth-first exploration strategy. This strategy explores the IOGG and notifies the remote strategy when there are no more rule transitions to explore. The symbolic strategy builds the IOSTS in Java objects using the explored rule transitions. The class diagram of the IOSTS is given in section 4.2.2. The remote strategy sends the IOSTS in JSON format as a HTTP POST request to the interface at ATM.

The breadth-first exploration strategy and the symbolic strategy are loosely coupled, such that other strategies can also be used if desired.

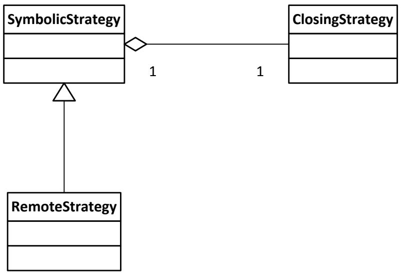


Figure 4.2: The class diagram of the exploration strategy interface

4.2.2 STSs

Figure 4.3 shows the class diagram of the IOSTS in GRATiS. The IOSTS is composed of Location, SwitchRelation, Gate, Interaction- and LocationVariable classes. A Location object can be the start and target of any number of SwitchRelation objects. A SwitchRelation object has two Location objects; the start and target location. It also has one Gate object, which can belong to any number of SwitchRelation objects. A Gate object can have any number of InteractionVariable objects, but an InteractionVariable object belongs to only one Gate object. The IOSTS has a

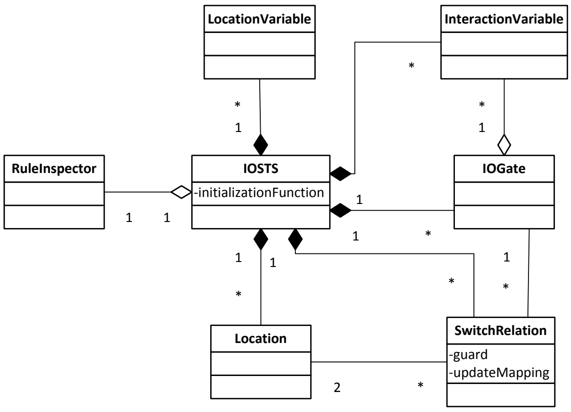


Figure 4.3: The class diagram of the IOSTS in GRATiS

singleton class, the RuleInspector, which contains the functionality of building guards and updates from rule graphs.

Figure 4.4 shows the object relations in accordance with the class diagram for the IOSTS in Figure 2.4. Note that the links between the *BoardGame* object and the Location, SwitchRelation, IOGate and InteractionVariable classes are not drawn for the sake of clarity.

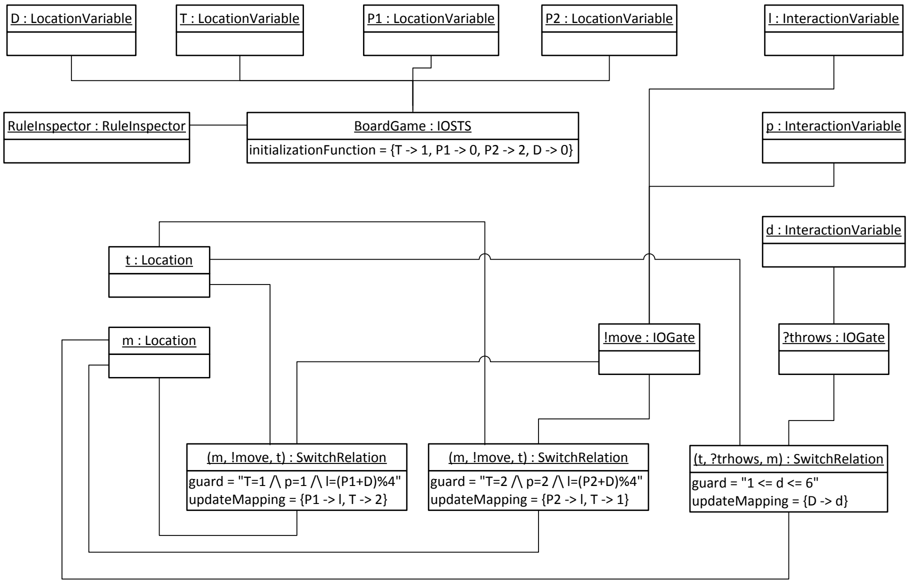

D : LocationVariable
T : LocationVariable
P1 : LocationVariable
P2 : LocationVariable
I : InteractionVariable
RuleInspector : RuleInspector
BoardGame : IOSTS
initializationFunction = {T -> 1, P1 -> 0, P2 -> 2, D -> 0}
p : InteractionVariable
d : InteractionVariable
t : Location
m : Location
!move : IOGate
?throws : IOGate
(m, !move, t) : SwitchRelation
guard = "T=1 /\ p=1 /\ I=(P1+D)%4"
updateMapping = {P1 -> I, T -> 2}
(m, !move, t) : SwitchRelation
guard = "T=2 /\ p=2 /\ I=(P2+D)%4"
updateMapping = {P2 -> I, T -> 1}
(t, ?trhows, m) : SwitchRelation
guard = "1 <= d <= 6"
updateMapping = {D -> d}

Figure 4.4: The object diagram of the IOSTS in Figure 2.4

4.2.3 ATM Interface

The ATM interface is one component in the Ruby on Rails framework. The interface is a controller in this Model-View-Controller framework. Controllers handle the HTTP requests given by the framework. The interface receives the IOSTS POST request, builds the IOSTS as Ruby objects and initiates the test using this IOSTS as model.

Chapter 5

Validation

This chapter covers the validation of GRATiS. The validation is done through examples and a case-study. Possible measurements on models and the performance are investigated in section 5.1. The examples, case-study and models are given in section 5.2.

5.1 Measurements

This section lists the possible measurements on the models and execution of GRATiS on those models.

5.1.1 Simulation and redundancy

For each of the cases, we can create IOSTSs in two ways: manually and through GRATiS. The IOSTS created by GRATiS and the IOSTS created by hand can be compared. It can be observed whether the IOSTS created by GRATiS simulates the IOSTS created by hand and vice versa. When either is not the case, the models show a different possible behaviour for the SUT. The reasons behind this difference are then explored.

It is possible that the generated IOSTS s is larger than the IOSTS built by hand t, even if both simulate each other. If t simulates s, it is measured whether the set of switch relations or location variables is strictly smaller in t than in s, while the other set is at least as small in t as in s. Formally:

$\mathit{traces}(s) \subseteq \mathit{traces}(t) \land ((|D_t| < |D_s| \land |\mathcal{L}_t| \leq |\mathcal{L}_s|) \lor (|D_t| \leq |D_s| \land |\mathcal{L}_t| < |\mathcal{L}_s|)$

This indicates the the IOSTS s has redundancy.

5.1.2 Performance

The performance in terms of runtime and heap-size can be measured and compared. Assuming both the IOSTS created by GRATiS and by hand simulate each other, these metrics will be the same for the testing part. Therefore, the runtime and heap size of the IOSTS creation is measured.

5.1.3 Model complexity

A good way of testing the complexity of the IOSTS and the IOGG is to hold an extensive social experiment, where groups create and maintain models in either formalism. However, this is out of the scope for this report. More is written about this social experiment in the Future Work section ??.

Instead of a social experiment, we investigated software complexity measurements in the literature and tested their applicability for IOGGs and IOSTSs.

Halstead's software science The complexity of the generated IOSTS and the IOGG can be measured using Halstead's software science [7]. This method is used in measuring software complexity and the prediction of faults. However, it can also be used in analyzing model complexity. In Halstead's software science, the operators and operands in the program code are counted. The operators are the function symbols, the operands are the identifiers. However, in order to apply these concepts to our setting, we have to identify what we consider to be operators and operands.

IOSTSs and IOGGs both have identifiers and function symbols. However, they also have nodes and edges. In an IOSTS, the locations are counted as nodes, the switch relations as edges. Nodes and edges are considered to be operands. In GROOVE, colors indicate a restriction or node/edge removal/creation. The node and edge colors are therefore considered as operators.

The distinct number of operators (n_1) and operands (n_2) are counted as well as the total number of operator occurrences (N_1) and operand occurrences (N_2). These metrics combined lead to the *Volume* (V) of the models.

Definition 5.1.1. Halstead's Volume (V) function
$(N_1 + N_2) * log_2(n_1 + n_2)$

In Halstead's software science, the volume of a program and a constant for the number of faults per volume unit are combined to give the expected faults in a program. Therefore, comparing the volumes of the IOSTS and IOGG gives an indication of the relative model complexity.

Cyclomatic Complexity COMMENT Ik heb intensief emails gewisseld met Lodewijk Bergmans over de toepassing van cyclomatic complexity, maar dit lijkt niet geschikt hier. Ik zal het wel behandelen /COMMENT

System Complexity COMMENT mooie, simpele metric voor OO. Niet toepasbaar hier, want onmogelijk om modules te definieren /COMMENT

5.1.4 Extendability

The models can be extended to include more functionality. In this measurement, a realistic scenario is introduced where additional functionality is required. It is then measured how much the complexity increases, using the measurment in section 5.1.3.

5.2 Models

This section shows the IOGG and IOSTS for each model example and the case study.

5.2.1 Example 1: boardgame

The boardgame is the running example of which the IOSTS and IOGG are given in Figures 2.4 and 2.20 respectively. In order to be consistent with the GG variable definition, the two ?*Player* and ?*Die* nodes receive the flags *var1*, *var2*, *var3* respectively, representing the variable labels.

5.2.2 Example 2: farmer-wolf-goat-cabbage

In this puzzle, a farmer, wolf, goat and cabbage are on one side of a river. The farmer can take upto one item to the other side. If the wolf and goat are on one side of the river without the farmer, the wolf eats the goat and the puzzle is reset. This also holds for the goat and the cabbage. The goal is to move all four to the other side of the river. The stimuli accepted by the puzzle are:

- $?n$ Move the farmer to the other river bank
- $?w$ Move the wolf to the other river bank
- $?g$ Move the goat to the other river bank
- $?c$ Move the cabbage to the other river bank

The responses given by the puzzle are:

- $!retry$ When $?w$, $?g$ or $?c$ is given but the farmer is not on the same river bank as the item
- $!eaten$ An item eats another item on one river bank, with the farmer on the other river bank
- $!done$ When all four items are on the other river bank

The IOGG of this puzzle is in Appendix A. The response rules '!retry', '!eaten' and '!done' have a higher priority. This ensures that a proper response is given after a move, before allowing more stimuli.

The IOSTS of this puzzle is is given in the formal definition, because it is too large for the visual representation. It uses boolean location variables to indicate whether the item is on the other side. These are $?N$, W, G, C for the farmer, wolf, goat and cabbage respectively. These are checked to see if an invalid move is done, an item is being eaten or the puzzle has been completed.

$$\begin{array}{lcl}
W &=& \{l_0, l_1\} \\
w_0 &=& l_0 \\
\mathcal{L} &=& \{N, W, G, C\} \\
\imath &=& \{N \mapsto false, W \mapsto false, G \mapsto false, C \mapsto false\} \\
\mathcal{I} &=& \{\} \\
\Lambda &=& \{?n, ?w, ?g, ?c, !eaten, !done, !retry\} \\
D &=& \{l_0 \xrightarrow{?n,\neg(N\neq G\wedge G=C)\vee(N\neq W\wedge W=G),\{N\mapsto\neg N\}} l_0, \\
&& l_0 \xrightarrow{?w,N=W\wedge(N=G\vee G\neq C)\wedge(N=W\vee W\neq G),\{W\mapsto\neg W,N\mapsto\neg N\}} l_0, \\
&& l_0 \xrightarrow{?g,N=G\wedge(N=G\vee G\neq C)\wedge(N=W\vee W\neq G),\{G\mapsto\neg G,N\mapsto\neg N\}} l_0, \\
&& l_0 \xrightarrow{?c,N=C\wedge(N=G\vee G\neq C)\wedge(N=W\vee W\neq G),\{C\mapsto\neg C,N\mapsto\neg N\}} l_0, \\
&& l_0 \xrightarrow{?w,N\neq W\wedge(N=G\vee G\neq C)\wedge(N=W\vee W\neq G),\{W\mapsto\neg W,N\mapsto\neg N\}} l_1, \\
&& l_0 \xrightarrow{?g,N\neq G\wedge(N=G\vee G\neq C)\wedge(N=W\vee W\neq G),\{G\mapsto\neg G,N\mapsto\neg N\}} l_1, \\
&& l_0 \xrightarrow{?c,N\neq C\wedge(N=G\vee G\neq C)\wedge(N=W\vee W\neq G),\{C\mapsto\neg C,N\mapsto\neg N\}} l_1, \\
&& l_1 \xrightarrow{!retry,true,\{\}} l_0, \\
&& l_0 \xrightarrow{!eaten,(N\neq G\wedge G=C)\vee(N\neq W\wedge W=G),\{N\mapsto false,W\mapsto false,G\mapsto false,C\mapsto false\}} l_0, \\
&& l_0 \xrightarrow{!done,N\wedge W\wedge G\wedge C,\{N\mapsto false,W\mapsto false,G\mapsto false,C\mapsto false\}} l_0\}
\end{array}$$

5.2.3 Example 3: bar tab system

This example models a bar tab system, where customers can order beer, wine and coke, costing \$1.50, \$2.10 and \$0.80 respectively. The price of the order adds to the customer's tab. Customers can pay their tab with money; they receive cash back if the payment exceeds the tab. The models are abstracted to include three customers. Furthermore, a customer can order only one drink. Drinks and payments are processed immediately before other drinks or payments can occur. The stimuli accepted by the system are:

- $?o(i, d)$ for ordering a drink $?d$ on bar tab $?i$
- $?p(i, p)$ for paying amount $?p$ on bar tab $?i$

The responses given by the system are:

- $!po(b)$ for processing an order giving the new bar tab balance $?b$
- $!pp(b, r)$ for processing a payment giving the new account balance $?b$ and the return funds $?r$

The IOGG of the bar tab system is in Appendix B. The '!po' and '!pp' rules have a higher priority than the '?o' and '?p' rules, to ensure an order is processed before a new one is made.

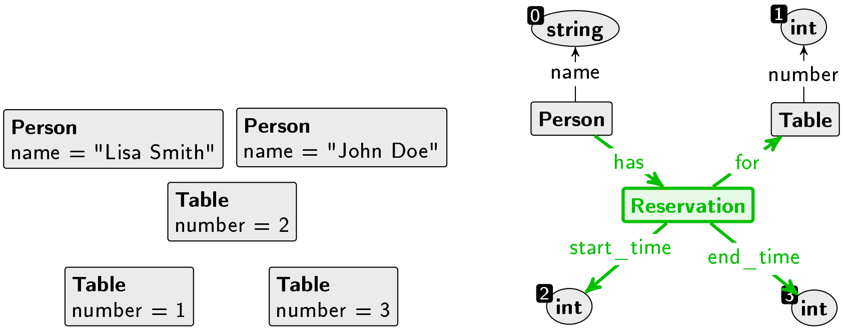
The formal definition for the IOSTS of the bar tab system is given below. The IOSTS uses the variables $?T_1, T_2, T_3$ to keep track of the bar tabs of the three people. It uses the variables $?I, P$ as temporary variables for the id and payment/price respectively. The function $?max$ takes the maximum value of its parameters.

$$
\begin{array}{lcl}
W & = & \{l_0, l_1, l_2\} \\
w_0 & = & l_0 \\
\mathcal{L} & = & \{T_1, T_2, T_3\} \\
\imath & = & \{T_1 \mapsto 0, T_2 \mapsto 0, T_3 \mapsto 0\} \\
\mathcal{I} & = & \{i, d, p, b, r\} \\
\Lambda & = & \{?o(i, d), ?p(i, p), !po(b), !pp(b, r)\} \\
D & = & \{l_0 \xrightarrow{?o(i,d),d=\text{`}coke'\wedge i\geq 1\wedge i\leq 3,\{I\mapsto i,P\mapsto 0.8\}} l_1, \\
 & & l_0 \xrightarrow{?o(i,d),d=\text{`}beer'\wedge i\geq 1\wedge i\leq 3,\{I\mapsto i,P\mapsto 1.5\}} l_1, \\
 & & l_0 \xrightarrow{?o(i,d),d=\text{`}wine'\wedge i\geq 1\wedge i\leq 3,\{I\mapsto i,P\mapsto 2.1\}} l_1, \\
 & & l_1 \xrightarrow{!po(b),I=1\wedge b=T_1+P,\{T_1\mapsto b\}} l_0, \\
 & & l_1 \xrightarrow{!po(b),I=2\wedge b=T_2+P,\{T_2\mapsto b\}} l_0, \\
 & & l_1 \xrightarrow{!po(b),I=3\wedge b=T_3+P,\{T_3\mapsto b\}} l_0, \\
 & & l_0 \xrightarrow{?p(i,p),i\geq 1\wedge i\leq 3,\{I\mapsto i,P\mapsto p\}} l_2, \\
 & & l_2 \xrightarrow{!pp(b,r),I=1\wedge b=max(T_1-P,0)\wedge r=max(-T_1+P,0),\{T_1\mapsto b\}} l_0, \\
 & & l_2 \xrightarrow{!pp(b,r),I=2\wedge b=max(T_2-P,0)\wedge r=max(-T_2+P,0),\{T_2\mapsto b\}} l_0, \\
 & & l_2 \xrightarrow{!pp(b,r),I=3\wedge b=max(T_3-P,0)\wedge r=max(-T_3+P,0),\{T_2\mapsto b\}} l_0\}
\end{array}
$$

5.2.4 Failed example: restaurant reservations

In this example, a restaurant with three tables allow customers to reserve tables for a certain time slot, given by a start and end time. The reserved time should be at least an hour. The models abstract again from the number of customers, setting it to two. The time span in which reservations can be made is a week. The maximum number of reservations is therefore bounded to $7 * 24 = 168$.

Figure 5.1a shows the initial graph of three tables at a restaurant and two potential customers. Figure 5.1b shows part of a rule that allows people to make reservations. The start and end times are timestamps represented by integers. This rule allows people to make multiple reservations. However, The reservations cannot be GG variables or this rule violates the unique GG variables constraint in section 3.4, because it has to assign a variable label to the ?*Reservation* node.



(a) The initial graph

(b) The make reservation rule

Figure 5.1: Part of the IOGG for the restaurant reservation system

Allowing a dynamic amount of reservations per person means that variables need to be introduced dynamically as well or more complex variables have to be used, such as arrays. This also holds for the IOSTS of this system. This example is to show the limitations on data and the abstractions that both models are required to make. Therefore this example is not included in the measurements.

5.2.5 Case study: Scanflow Cash Register Protocol

The system used for this case study is a *self-scan register*, which allows customers of a supermarket to scan and pay for their products without help of an employee. Figure 5.2 shows this self-scan register. The system contains a *scanflow unit*, which scans the products, and a *cash register*, which handles the payment. The communication protocol between the register and the scanflow unit is the system which is tested. The register is the SUT, the scanflow unit gives stimuli to and receives responses from the register. The code of the register is available in a simulator, which is used for the actual testing.

Figure 5.2: A self-scan register

Stimuli and responses Appendix C gives the most relevant parts of the specification document for this communication protocol. From this document, the following stimuli were designed:

?*SIGNON*	Requests the register to sign on
?*SIGNOFF*	Requests the register to sign off
?*OPEN*	Opens a new account
?*ARTREG*(*barcode*, *amount*)	Requests registration of an article, with *barcode* representing its key and *amount* representing the amount of articles to register
?*STAMPREG*(*amount*)	Requests registration of loyalty points, where *amount* is the amount of loyalty points to register
?*ARTID*(*barcode*)	Requests identification of an article, with *barcode* representing its key
?*ARTRET*(*barcode*, *amount*)	Returns an article where *barcode* is the article being returned and *amount* representing the amount of aricles to return
?*CLOSE*	Closes the account
?*ENDTOT*	Requests the end total of the registered articles
?*TRANS*(*trans_id*)	Inform the register that the payment was made, where *tr_method* represents the payment method, e.g. cash, credit card
?*RECEIPT*	Requests the register return the receipt.
?*PRINT*	Requests the register to print additional information on the receipt
?*IDLE*	Requests rounding-up of the account
?*RESUME*	Requests the register to continue operation from an error state
?*RHCOPY*	Requests the register to print a hardcopy of a receipt
?*RESETCR*	Requests the register to execute a reset
?*GET*(*var_name*)	Requests to get the value of a certain register variable represented by *var_name*

The SCRP responses follow a coding scheme, much like the definition for FTP. The responses are a 3-digit status code, where each digit has a special significance. The first digit denotes the essence of the related action:

- 2yz Ok - Command has been accepted and the action is successfully completed. A new command may be issued.
- 4yz Again - Command was not a accepted as a result of a temporary error condition. It is encouraged to request the same action again.
- 5yz Fail - Command was not accepted as a result of a permanent error condition. It is discouraged to request the same action again.

The following function categories are encoded in the second digit:

- x0z Syntax - Command contained a syntax error (50z), or default indication if no other function category applies (10z, 20z, 30z, 40z)
- x1z Information - Response related to access of information.
- x2z Connection - Response related to connection/service.
- x3z Account - Response related to account management.
- x4z Transaction - Response related to (payment) transaction.
- x5z Signing - Response related to sign on/off operations.
- x6z Printing - Response related to receipt printing.

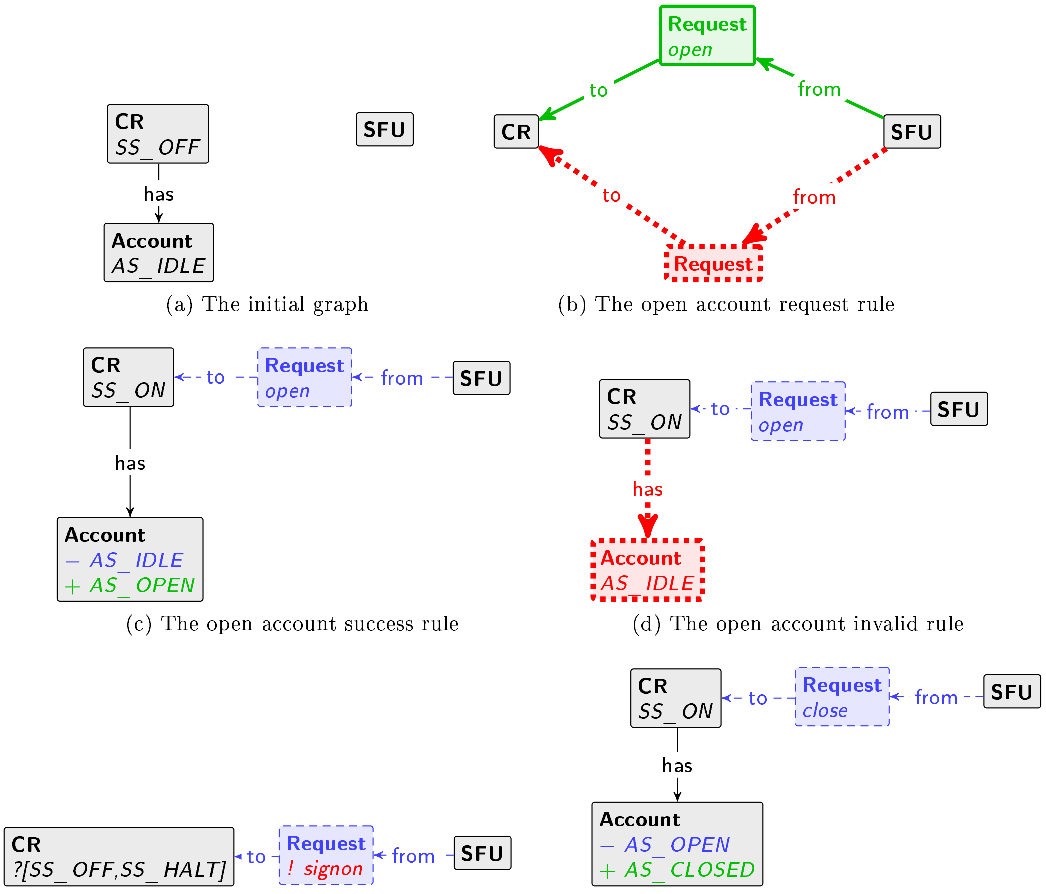
The third digit gives a finer gradation in each of the function categories, which leads to the complete list of responses:

!201	Resumed operation
!202	Cash Register restored
!210($cr\_variable, cr\_value$)	Gives the value of the variable $cr\_variable$
!212($description, price$)	Gives the description and price of a registered article
!213($description, price$)	Gives the description and price of a registered article
!220	SCRP Service ready
!221	SCRP Service terminating
!230($endtotal$)	Gives the end total for all registered articles.
!231	Account opened
!232($nr\_articles, subtotal$)	Article registered
!233	Account idled
!240	Transaction succeeded
!250	Signed Off
!251	Signed On
!260	Data printed
!261($html\_text$)	Gives the receipt as HTML
!450	Signing rejected
!500	Unknown command
!501	Syntax error
!502	Command failed
!503	Error state
!510	No such variable
!511	No such article
!512	No stable weight
!530	No such account
!531	Invalid account state
!540	No such transaction method
!541	Busy transacting
!542	Transaction failed
!550	Not signed on
!551	Authentication failed
!560	CR-printing inactive
!561	SFU-printing inactive

IOGG Some selected rules of the IOGG of this communication protocol are shown in Figure 5.3. In total the IOGG has 94 rules. They are all listed in Appendix D. Figure 5.3a shows the initial graph. The ?CR node is the cash register, the ?SFU is the scanflow unit. The first node has the flag SS_OFF representing that the register is off. There is one account which can be in states idle, open, closed and transing. When the customer places items on the belt, a new account is opened. Figure 5.3b shows the general request structure. As long as there is no request, a request can be sent. This rule requests the opening of an account. Figure 5.3c shows the rule for giving a success response. The request node is deleted such that new request nodes can be created again. The rule checks if an account is not already opened and opens an idle account. Figure 5.3d shows the rule for the error message received when an account is already opened.

For all these rules, the ?CR node has to have the flag SS_ON representing the register to be signed on. Figure 5.3e shows the response to a request when the register is not signed on. Apart from the signon request, no other request is allowed in this state. Figure 5.3f shows the rule closing the account.

IOSTS The IOSTS of the system as created by Axini is too large to show here. It has X switch relations and Y location variables.



(a) The initial graph

(b) The open account request rule

(c) The open account success rule

(d) The open account invalid rule

(e) The not signed on error rule

(f) The close account success rule

Figure 5.3: The IOGG of the Scanflow Cash Register Protocol

Bibliography

[1] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34(1):1 – 54, 1999.

[2] Axel Belinfante. JTorx: A tool for on-line model-driven test derivation and execution. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 266–270. Springer Berlin / Heidelberg, 2010.

[3] E. Brinksma. A theory for the derivation of tests. *Protocol Specification, Testing, and Verification VIII*, 1988.

[4] Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. STG: A symbolic test generation tool. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 151–173. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-46002-0_34.

[5] D.M. Cohen, S.R. Dalal, M.L. Fredman, and G.C. Patton. The aetg system: an approach to testing based on combinatorial design. *Software Engineering, IEEE Transactions on*, 23(7):437 –444, jul 1997.

[6] A. H. Ghamarian, M. J. de Mol, A. Rensink, E. Zambon, and M. V. Zimakova. Modelling and analysis using GROOVE. *International journal on software tools for technology transfer*, 14(1):15–40, February 2012.

[7] M.H. Halstead. *Elements of Software Science*. Elsevier, 1977.

[8] Hasan and Ural. Formal methods for test sequence generation. *Computer Communications*, 15(5):311 – 325, 1992.

[9] R. Heckel. Graph transformation in a nutshell. *Electronic Notes in Theoretical Computer Science*, 148(1 SPEC. ISS.):187–198, 2006.

[10] M.R.A. Huth and M. Ryan. *Logic in computer science: modelling and reasoning about systems*. Cambridge University Press, 2000.

[11] C. Laird, L. & Brennan. *Software measurement and estimation: A practical approach*. IEEE Computer Society/Wiley, 2006.

[12] Tim A.C. Willemse Lars Frantzen, Jan Tretmans. Test generation based of symbolic specifcations. Technical report, Nijmegen Institute for Computing and Information Sciences (NIII), 2005.

[13] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090 –1123, aug 1996.

[14] R. Busser M. Blackburn and A. Nauman. Why model-based test automation is different and what you should know to get started. *International Conference on Practical Software Quality and Testing*, 2004.

[15] Joost-Pieter Katoen Manfred Broy, Bengt Jonsson and Alexander Pretschner. *Model-Based Testing of Reactive Systems.* Springer-Verlag Berlin Heidelberg, 2005.

[16] Thomas J. McCabe. Structured testing: A software testing methodology using the cyclomatic complexity metric. *National Bureau of Standards, Special Publication*, 1982.

[17] Steve McConnell. Software quality at top speed. *Softw. Dev.*, 4:38–42, August 1996.

[18] Lev Nachmanson, Margus Veanes, Wolfram Schulte, Nikolai Tillmann, and Wolfgang Grieskamp. Optimal strategies for testing nondeterministic systems. *SIGSOFT Softw. Eng. Notes*, 29:55–64, July 2004.

[19] R. De Nicola and M.C.B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1-2):83 – 133, 1984.

[20] Martin Pol. *Testen volgens Tmap (in dutch, Testing according to Tmap).* Uitgeverij Tutein Nolthenius, 1995.

[21] S.J. Prowell. JUMBL: a tool for model-based statistical testing. In *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*, page 9 pp., jan. 2003.

[22] A. Rensink. Towards model checking graph grammars. In S. Gruner and S. L. Presti, editors, *Workshop on Automated Verification of Critical Systems (AVoCS), Southampton, UK*, volume DSSE-TR-2003-02 of *Technical Report*, pages 150–160, Southampton, 2003. University of Southampton.

[23] A. Rensink. The GROOVE simulator: A tool for state space generation. In J. L. Pfaltz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485, Berlin, 2004. Springer Verlag.

[24] Vlad Rusu, Lydie du Bousquet, and Thierry Jéron. An approach to symbolic test generation. In Wolfgang Grieskamp, Thomas Santen, and Bill Stoddart, editors, *Integrated Formal Methods*, volume 1945 of *Lecture Notes in Computer Science*, pages 338–357. Springer Berlin / Heidelberg, 2000. 10.1007/3-540-40911-4_20.

[25] G. J. Tretmans and H. Brinksma. Torx: Automated model-based testing. In A. Hartman and K. Dussa-Ziegler, editors, *First European Conference on Model-Driven Software Engineering, Nuremberg, Germany*, pages 31–43, December 2003.

[26] Jan Tretmans. *A Formal Approach to Conformance Testing.* PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[27] Jan Tretmans. Test generation with inputs, outputs, and quiescence. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 127–146. Springer Berlin / Heidelberg, 1996. 10.1007/3-540-61042-1_42.

[28] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, pages n/a–n/a, 2011.

[29] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-based testing of object-oriented reactive systems with spec explorer. In Robert Hierons, Jonathan Bowen, and Mark Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 39–76. Springer Berlin / Heidelberg, 2008.

[30] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, December 1997.

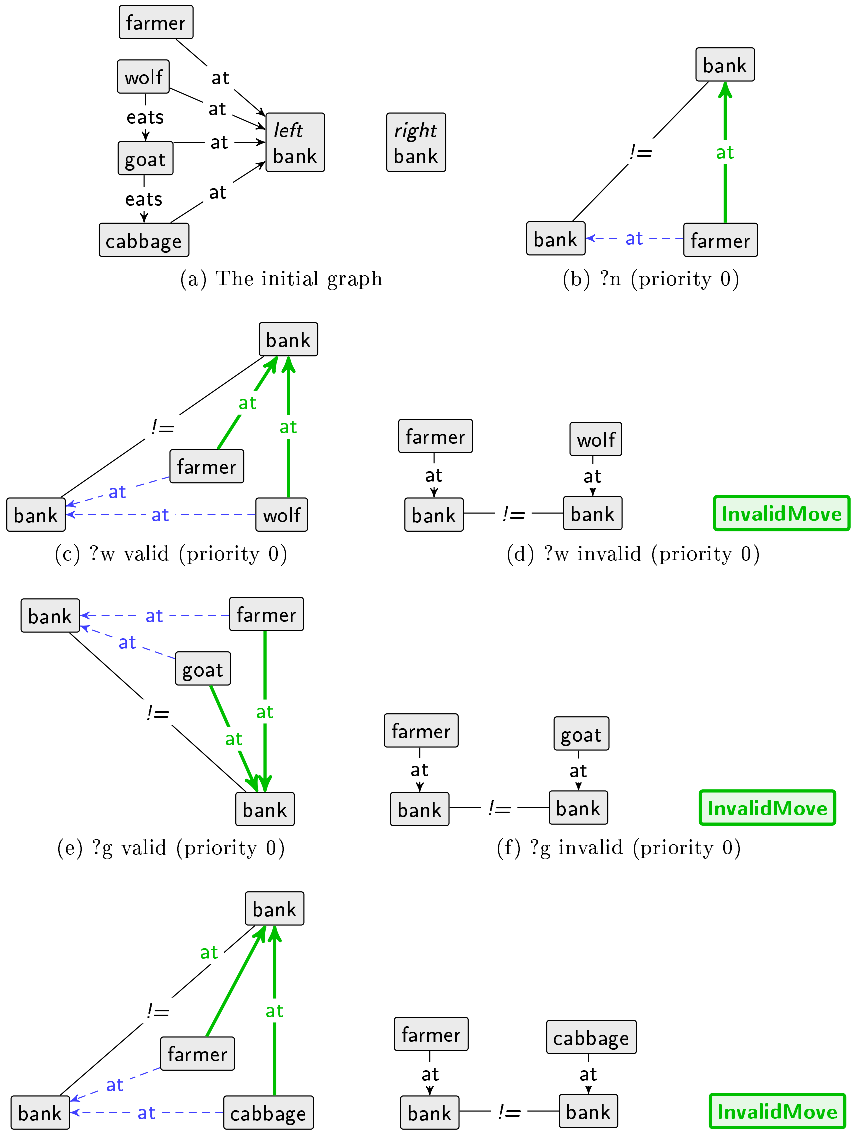
List of Symbols

d	A switch relation	11
f	A function symbol	10
l	A label	8
l_v	A value label	25
l_a	A variable label	25
m	A graph transfomration rule match on a graph	15
q_0	An initial state	8
r	A graph transformation rule	15
s	A sort	10
t	A transition	8
u	A rule transition	16
v	A variable	10
w_0	An initial location	11
ι	An I/O type	9
z_a	A variable anchor	25
D	Set of switch relations	11
F	Set of function symbols	10
G	A graph	14
G_0	An initial graph	15
H	A rule graph	14
L	Set of labels	8
L	Set of input-output labels	16
M	Set of graph transfomration rule matches	15
Q	Set of states	8
R	Set of graph transformation rules	15
R_Y	Set of input-output graph transformation rules	16
S	Set of sorts	10
T	Set of transitions	8
T_Y	Set of I/O transitions	9
W	Set of locations	11
U	Set of rule transitions	16
U_Y	Set of input-output rule transitions	16
Y	Set of I/O types	9
$\mathbb{U}$	A universe	10
$\mathbb{V}$	The universe of graph nodes	14
$\mathbb{E}$	The universe of graph edges	14
$\mathbb{W}$	The universe of standard graph nodes	14
$\imath$	Term mapping for location variable initialization	11
$\mathcal{A}$	An algebra	10
$\mathcal{B}$	Set of terms with boolean type	10
$\mathcal{G}$	Set of graphs	16

$\mathcal{I}$	Set of interaction variables	11
$\mathcal{L}$	Set of location variables	11
$\mathcal{P}$	A point algebra	10
$\mathcal{T}$	Set of terms	10
$\mathcal{V}$	Set of variables	10
γ	Guard of a switch relation	11
ϕ	A function	10
λ	A gate of a switch relation	11
μ	Term-mapping function	10
ν	Valuation function	10
ρ	Update mapping of a switch relation	11
Φ	Set of functions	10
Λ	Set of gates	11
Λ_Y	Set of I/O gates	12

Appendix A

Farmer-Wolf-Goat-Cabbage IOGG



(a) The initial graph

(b) ?n (priority 0)

(c) ?w valid (priority 0)

(d) ?w invalid (priority 0)

(e) ?g valid (priority 0)

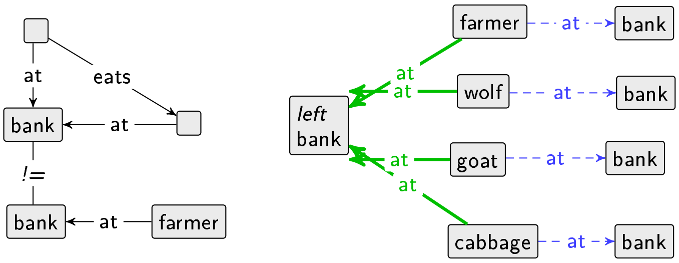
(f) ?g invalid (priority 0)

(g) ?c valid (priority 0)

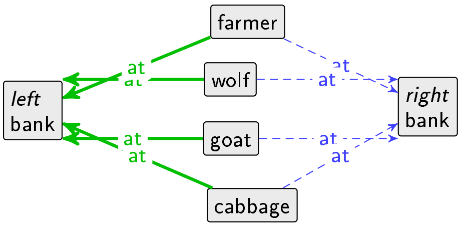
(h) ?c invalid (priority 0)

InvalidMove

(i) !retry (priority 1)



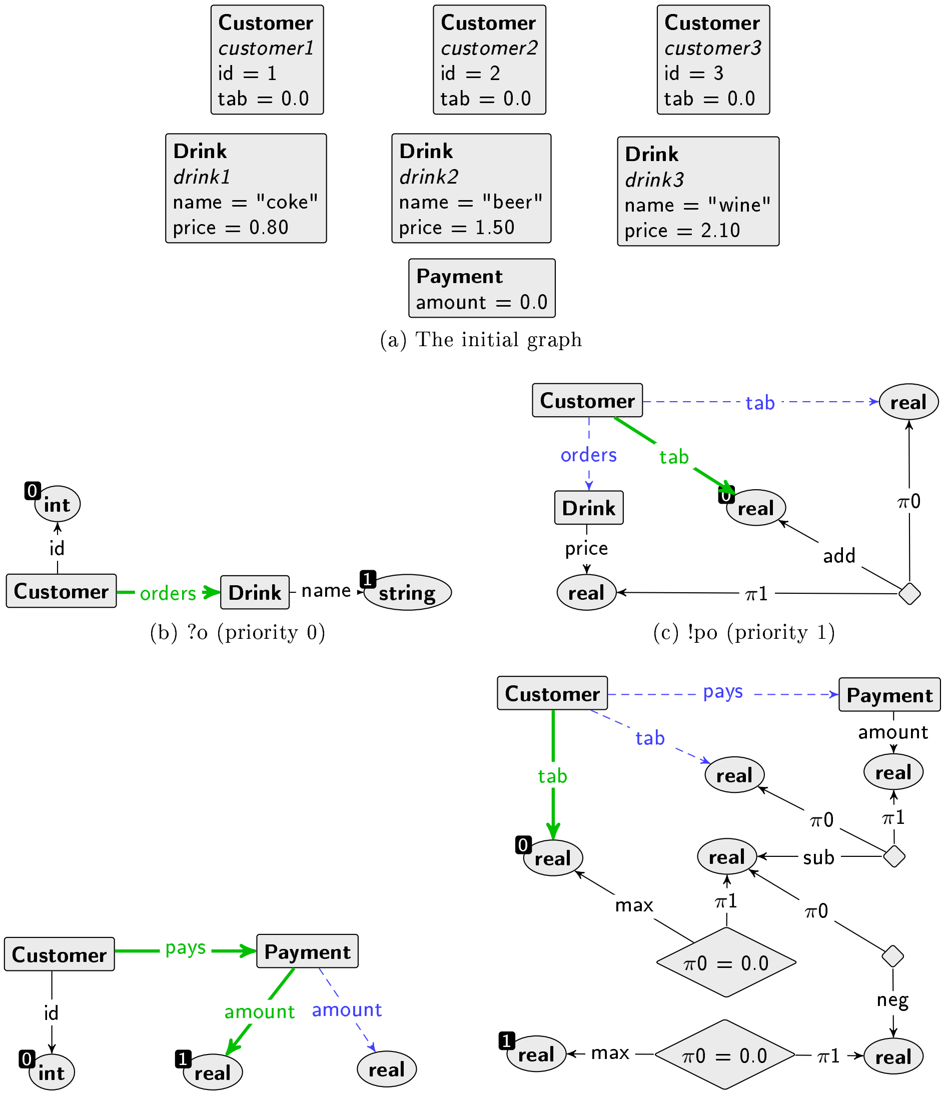
(j) !eaten (priority 1)



(k) !done (priority 1)

Appendix B

Bar Tab IOGG



(a) The initial graph

(b) ?o (priority 0)

(c) !po (priority 1)

(d) ?p (priority 0)

(e) !pp (priority 1)

Appendix C

SCRP Commands & Responses

Registration Accounts

The SCRP protocol incorporates the concept of an account, maintained in the CR on behalf of the SFU, onto which articles can be registered. Registered articles form the basis for the amount for which the customer is charged. A registration account can be in state AS_IDLE, AS_OPEN, AS_CLOSED, AS_TRANSING or AS_ENDING, see Figure C.1.

When the CR service is started, the account is typically initialized to AS_IDLE. An OPEN command issued by the SFU opens the account for registration. Registration is only possible in this state. After registration is finished, the CLOSE command must be invoked. In the AS_CLOSED state, a TRANS command initiates a (payment) transaction. This command must be re-issued until the transaction succeeds. A successful transaction causes the account to enter the AS_ENDING. In this state, the IDLE command makes the account AS_IDLE, i.e. available to be opened for the next session.

General Responses

The following responses can, when appropriate, be generated as a response on any command.

500 Unknown command	In case the command is not recognized.
501 Syntax error	In case the command contains a syntax error (e.g. mandatory argument is missing).
502 Command failed	As a last resort, when no other appropriate response can be given.
503 Error state	In case the CR is in Error state and the requested command cannot be processed due to that.
504 Weighing not available	This response should be used in the case a Cash register has a scale configured, but no connection between the Cash register and the scale is established, the Cash register must go to an error state and send a 504 error (weighing not available).
550 Not signed on	If the CR requires to be signed on for the command involved.

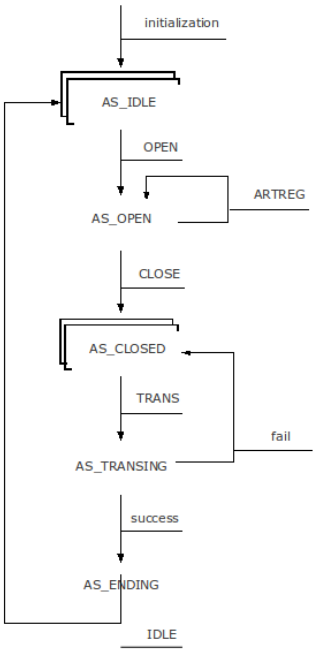


Figure C.1: Registration Account State Machine

Commands

Initialize SCRP Service

Command:	None; a transport connection suffices.
Description:	Initializes the SCRP service on the CR. The SFU must synchronize to the CR after the connection has been established.
Responses:	220 SCRP Service ready

Terminate SCRP Service

Command:	QUIT
Description:	Requests the CR to smoothly terminate the SCRP service. Although the SFU should wait for all accounts to be AS_IDLE before issuing this request, the CR must honour it in any state. Any non-IDLE account will be processed by the CR in an implementation defined manner. The CR remains ready to accept a (new) SFU connection.
Responses:	221 SCRP Service terminating

Cash Register Sign On

Command:	SIGNON <SFU_id>[:<password>]
Description:	Requests the CR to sign on, making the actual signing state SS_ON. The CR is allowed to reject the request if the current signing state is SS_HALT.
Responses:	251 Signed On 450 Signing rejected 551 Authentication failed

Cash Register Sign Off

Command: SIGNOFF

Description: Requests the CR to sign off, making the actual signing state SS_OFF or SS_HALT. The CR is allowed to reject the request if not all accounts are in AS_IDLE state. Otherwise, any non-IDLE account will be processed by the CR in an implementation defined manner.

Responses: 250 Signed Off
450 Signing rejected

Get CR Variable

Command: GET <cr_variable>

Description: Requests to get the value of a certain CR Variable. The CR must honour this request in any state of the system. Possible variables and their values are listed in Table 3.

Responses: 210 <cr_variable>:<cr_value>
510 No such variable

Open Account

Command: OPEN [<account>|<barcode>]

Description: Opens an account for subsequent article registrations. The CR is requested to perform this action before registration of articles commences. If the CR implementation is such that the account state is “default opened" after the previous payment sequence, the command may internally have no effect but a positive response is still required. A positive response must report the identity of the account opened. If <account> or <barcode> is specified (both may be used alternatively), the command requests to recall the associated stored account. The format of <barcode> is the same as used by the CR to produce a store/recall ticket. A preferred implementation of this command returns an account number that is identical to the requested <account> (or identical to the account number from which <barcode> was constructed). Returning another account number is discouraged but allowed. The newly opened account must be put in state AS_OPEN and must inherit all account data from the account that is recalled, including the state specific context.

Responses: 231 <account> Account opened
530 No such account
531 Invalid account state

Close Current Account

Command: CLOSE

Description: Closes the current account for further registration. The CR is requested to perform this action before a transaction sequence is initiated. A positive response must include the total amount involved. The response may be preceeded by a series of ‘display responses’, explaining the calculation from the previous subtotal to the current endtotal (e.g. mix-match calculation results).

Responses: { 212 <description>[:<price>[:<amount>]] }
230 <endtotal> Account closed
531 Invalid account state

Request Article Identification

Command: ARTID <barcode>

Description: Requests identification of an article, with <barcode> representing its key. This command is not a request to register the article. The CR must honor this request in any account state. A positive response must be given by means of a 211 response or a 213 response, their usage is alternative, which means that is up to the CR to use one or the other to respond to a given ARTID command. However systems using the certified weighing capability of the Cash Register must use the 213 response. Compared to the 211 response, the 213 response allows the CR to provide a full-scale specification of the attributes that characterize an article. A detailed definition of these responses including the error responses, is given hereunder. The length of the responses may never be larger then 512 characters.

Responses: 213

Request Article Registration

Command: ARTREG [<barcode>[:<amount>]]

Description: Requests registration of an article, with <barcode> representing its key and <amount> representing the amount of articles to register. This request is not issued before an account has been opened successfully. A positive response must consist of the following elements: <nr_articles> Actual number of articles registered, <subtotal> Actual subtotal of articles registered. The response must be preceded by a series of ‘display responses’ with all data related to the registration of this article (e.g. linked articles, discounts, etc.). If <barcode> is omitted, the command requests the complete list (as a series of ‘display responses’) of articles registered in the current account, which might be a recalled account. In case of a recalled account for which a partial payment was carried out on the originating (stored) account, the list of ‘display responses’ must include descriptive lines explaining the results of the partial payment(s) done; all in consistence with the value and definition of subtotal. If <amount> is omitted, it will default to 1.

Responses: { 212 <description>[:<price>[:<amount>]] }
232 <nr_articles>:<subtotal> Article registered
511 No such article
531 Invalid account state

Request Account Transaction

Command: TRANS <tr_method»[:<amount>]

Description: Requests transaction of the ‘oldest’ account from repository state AS_CLOSED. The account must be closed before the request can be honoured. For details regarding transaction methods. A negative response (failure) must include an indicator that specifies the cause of the failure. Table 8 lists an overview of possible failures for each transaction method. Both transaction method and failure indication must be represented as ASCII strings and interpreted in a case-insensitive manner. If <amount> is omitted, the actual endtotal is assumed implicitly. Otherwise, <amount> indicates the requested transaction amount. The format of the <amount> argument is identical to that of <price>. The semantics of <amount> in relation to transaction methods TM_STORE and TM_FLUSH is undefined; in case <amount> is specified for these transaction methods, the response must report Failure Indication FI_AMOUNT (see table below). After a successful transaction, the remaining endtotal must be updated accordingly and in correspondence with the applicable exchange calculation rules that reside in the CR; this endtotal value is also returned in the (new) “240"-response. The resulting endtotal is by definition equal to 0 if the transaction was successful and the <amount> argument was omitted, even for TM_STORE and TM_FLUSH. Different from what is specified in the Account State Machine of Figure C.1 the account state remains AS_CLOSED after a successful transaction while the resulting endtotal is unequal to 0. This allows for repeated/split TRANS requests.

Responses: 240 Transaction succeeded
531 Invalid account state
540 No such transaction method
541 Busy transacting
542 <failure_ind> Transaction failed

Print to Receipt (CR-Printing only)

Command: PRINT <account>:<html_text>

Description: Requests the CR to print HTML-style data to the receipt of the specified account.

Responses: 260 Data printed
530 No such account
531 Invalid account state
560 CR-printing inactive

Query Receipt (SFU-Printing only)

Command: RECEIPT

Description: Queries the CR to generate a multiline response with the entire receipt of the account currently in AS_ENDING state.

Responses: { 261 <html_text> }
531 Invalid account state
561 SFU-printing inactive

Round-up Account

Command: IDLE
Description: Requests rounding-up of the account in AS_ENDING state. A successful response makes the account state AS_IDLE. In case of CR-printing, this command may cause he CR to activate the receipt cutter. With SFU-printing, this is the acknowledge to the CR that the receipt has been correctly printed.
Responses: 233 Account idled
531 Invalid account state

Resume Operation

Command: RESUME
Description: Requests the CR to continue operation from Error state. Once the CR is in Error state, this command is needed to resume normal operation.
Responses: 201 Resumed operation
503 Error state

Request Account EndTotal

Command: ENDTOT
Description: Requests to calculate the actual endtotal for the current account. The request is only valid in state AS_OPEN. The response must be preceded by a series of ‘display responses’, explaining the difference between the current subtotal and the calculated endtotal. As such, the responses to this command are similar to those of the “CLOSE" command. In contrast with the “CLOSE" command, “ENDTOT" must not affect the account state and may be issued repeatedly.
Responses: { 212 <description>[:<price>[:<amount>]] }
230 <endtotal> Account endtotal
531 Invalid account state

Request print of Receipt Hardcopy

Command: RHCOPY [<account>|<barcode>]
Description: Requests the CR to print a hardcopy of a receipt. If <account> or <barcode> is omitted, requests to print a hardcopy of the last printed receipt. This function is mandatory. A preferred implementation, however, must also support the arguments <account> or <barcode> (both may be used alternatively), in which case the command requests to print a hardcopy of the receipt associated to the specified account. The format of <barcode> is syntactically the same as used by the CR to produce a store/recall ticket. It is preferred to accept this command in all/most account states recognized by the SCRP protocol. Accepting this command in account state AS_IDLE is required, though. The receipt hardcopy must consist of all CR-generated print data for the designated account, as well as all data that was PRINT-ed for that account on behalf of the SFU.
Responses: 260 Data printed
530 No such account
531 Invalid account state
560 CR-printing inactive

Cash Register Reset

Command: RESETCR

Description: Requests the CR to execute a reset. The function will only be possible to execute manually from within the SFU.

Responses: 202 Cash Register restored
531 Invalid account state

Certified Weighing

Command: GET CERTDATA <barcode>

Description: This command requests certified weight data from the CR to be presented on the screen for human verification. The format of <barcode> is missing the embedded weight or price information. The Command can only be issued during the state AS_OPEN.

Responses: 214
512 No stable weight
531 Invalid account state

Appendix D

Scanflow Cash Register Models